



Irene Weber

VBA für Office-Automatisierung und Digitalisierung



Springer Vieweg

VBA für Office-Automatisierung und Digitalisierung

Irene Weber

VBA für Office- Automatisierung und Digitalisierung

Irene Weber
Hochschule für Angewandte Wissenschaften
Kempten
Kempten, Deutschland

ISBN 978-3-658-42716-0 ISBN 978-3-658-42717-7 (eBook)
<https://doi.org/10.1007/978-3-658-42717-7>

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://portal.dnb.de> abrufbar.

© Der/die Herausgeber bzw. der/die Autor(en), exklusiv lizenziert an Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2024

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von allgemein beschreibenden Bezeichnungen, Marken, Unternehmensnamen etc. in diesem Werk bedeutet nicht, dass diese frei durch jedermann benutzt werden dürfen. Die Berechtigung zur Benutzung unterliegt, auch ohne gesonderten Hinweis hierzu, den Regeln des Markenrechts. Die Rechte des jeweiligen Zeicheninhabers sind zu beachten.

Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Planung/Lektorat: Petra Steinmueller
Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature.

Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Das Papier dieses Produkts ist recycelbar.

Vorwort

Den Anlass dafür, dieses Buch zu schreiben, gaben meine VBA-Kurse in der Fakultät Maschinenbau an der Hochschule für Angewandte Wissenschaften Kempten. Als ich VBA in das Lehrprogramm aufnahm, war das Interesse an Programmierung und Informatik-Themen allgemein in der Studierendenschaft noch nicht besonders groß und viele Studierende befassten sich in der Lehrveranstaltung nur widerstrebend damit. Wenige Semester später, als die ersten aus den Industriepraktika zurückkehrten, war das Feedback dann sehr positiv. Schon mit den Grundkenntnissen aus einem einsemestrigen VBA-Kurs waren Studierende in der Lage, wertvolle Beiträge im Unternehmen zu leisten. Seitdem haben viele unserer Studierenden VBA in Studienprojekten, Industriepraktika, Abschlussarbeiten und als Werkstudenten erfolgreich eingesetzt und Projekte damit realisiert. Diese Erfahrungen zeigen den praktischen Nutzen, den VBA in der Industrie hat. Sowohl die Erfolge wie auch die Schwierigkeiten, die in den Berichten von Studierenden zum Ausdruck kamen, haben zur Themenauswahl dieses Buchs angeregt.

Einige Studierende vertiefen ihre VBA-Kenntnisse in einem fortgeschrittenen Kurs. Hier entwickeln sie Lösungen für echte Anwendungssituationen, die sie aus der Praxis als Werkstudierende, Selbständige und oft auch aus dem Vereinshrenamt mitbringen. So entstehen nützliche, auf die jeweiligen Aufgaben spezialisierte Verwaltungstools und Automatisierungen. Andere befassen sich aus persönlichem Interesse mit innovativen und experimentellen Aufgabenstellungen. Um einige Beispiele nennen: J. Schädler hat die CAD-Software CATIA automatisiert. M. Hoffmann steuert seine automatische Hobby-Bierbrauanlage aus einem in Excel realisierten Rezeptkonfigurator, der ein Ablaufprogramm generiert und außerdem Rezepte, Vorräte und Bierbewertungen verwaltet und Flaschenetiketten druckt. S. Benischke und M. Lankau haben ein legendär schönes Brettspiel mit exzellenter Usability auf Excel-Basis realisiert. Ein Routenplaner, der die OpenRouteService-API verwendet und auch Fahrtkosten berechnet, stammt von A. Zipfel. J. Raffler hat experimentell die Möglichkeiten und Grenzen der Integration von Excel und Python evaluiert. L. Grotz hat ein professionelles Tool zur Berechnung der nötigen Kollektorfläche für die Beheizung von Freischwimmhäuden entwickelt, das die monatlichen Durchschnittstemperaturen für den geplanten Pool-Standort und Globalstrahlungsdaten aus Webdiensten importiert. E. Altun hat das Tool „Digitaler Urlaubsantrag“[®]

eingebracht. Mit der Nutzung der PayPal-API aus VBA hat sich M. Nakov befasst. V. Koffler ist es gelungen, eine für Experimente benötigte, über USB angeschlossene Konstantstromquelle per VBA zu konfigurieren, Rückgabewerte auszulesen und diese in einem Exceldiagramm in Echtzeit zu visualisieren. T. Wolfram und M. Heitzer haben sich daran gemacht, zu einem Reiseziel und Erlebniswünschen per VBA eine Broschüre mit einem Reisevorschlag zu generieren. Der SOLIDWORKS Schraubengenerator von M. Probst erzeugt aus einem Satz Parameter auf Knopfdruck ein normgerechtes Modell einer Schraube. Y. Üzüm hat gezeigt, wie gut eine auf einem Excel-Arbeitsblatt gestaltete Bedienoberfläche aussehen und funktionieren kann. Danke für Euer Engagement und Euren Ideenreichtum! Mit diesem Buch möchte ich einen Beitrag dazu leisten, dass weitere solche Projekte schnell und mit wenig Mühe in guter Qualität verwirklicht werden.

Gosheim
im Dezember 2023

Irene Weber

MARKEN/WARENZEICHEN DRITTER

Microsoft, Microsoft Office, Microsoft Word, Microsoft PowerPoint, Microsoft Excel, Microsoft Outlook sind Warenzeichen oder eingetragene Warenzeichen der Microsoft Corporation. SAP® ist eine Marke oder eingetragene Marke der SAP SE oder ihrer verbundenen Unternehmen in Deutschland und anderen Ländern. SOLIDWORKS™ ist eine Marke von Dassault Systèmes oder ihrer verbundenen Unternehmen in Frankreich und in anderen Ländern.

Inhaltsverzeichnis

1	Einleitung	1
2	Einführung in VBA	5
2.1	Die Programmiersprache VBA	6
2.2	VBA-Programme entwickeln und ausführen	8
2.3	Variablen, Konstanten und Wertzuweisung	15
2.4	Operatoren	21
2.5	Schleifen	25
2.6	Verzweigungen	30
2.7	Prozeduren und Funktionen	32
2.8	Module und Sichtbarkeit von Variablen und Prozeduren	38
2.9	Fazit	42
3	Objektbasiertes Programmieren	43
3.1	Klasse und Objekt	44
3.2	Der Objekttyp Collection	45
3.3	Befehle und Schreibweisen zum Umgang mit Objekten	46
3.4	Objekte aus dem Objektmodell von Excel	47
3.5	Demo-Anwendung „Verlinkte Datenblätter“	52
3.6	Fazit und Ausblick	60
4	Basis-Techniken	61
4.1	Fehlerbehandlung	62
4.2	Objektbibliotheken einbinden und nutzen	71
4.3	Fazit und Ausblick	78
	Literatur	78
5	Dateiverzeichnissystem	79
5.1	Überblick über das Objektmodell des Dateiverzeichnissystems	80
5.2	Code-Beispiele	80

5.3	Demo-Anwendung „Dokumente-Massenbearbeitung“	83
5.4	Code-Beispiel „Log-Datei“	85
5.5	Fazit	87
6	Word	89
6.1	Das Objektmodell von Word	90
6.2	Demo-Anwendung „Word aus Excel“	96
6.3	Textmarken und Tabellen	102
6.4	Demo-Anwendung „Produktblätter“	103
6.5	Dokumente automatisiert bearbeiten mit Suchen und Ersetzen	106
6.6	Demo-Anwendung „Etiketten-Tool“	110
6.7	Fazit	115
	Literatur	115
7	PowerPoint	117
7.1	Einführung in das Objektmodell von PowerPoint	118
7.2	Demo-Anwendung „PowerPoint-Generator“	122
7.3	Fazit	129
	Literatur	129
8	Outlook I – Elemente automatisch bearbeiten	131
8.1	Anmeldung in Outlook	132
8.2	Elemente des Outlook-Objektmodells I: Ordner	133
8.3	Objekttypen für Elemente	138
8.4	Demo-Anwendung „Posteingangsverwaltung“	139
8.5	Demo-Anwendung „Anhänger-verwaltung“ mit komplexen Filtern	141
8.6	Demo-Anwendung „Termine-Controlling“	146
8.7	Demo-Anwendung „Mailing-Tool“	148
8.8	Fazit und Ausblick	152
	Literatur	152
9	Outlook II und MS Forms – Interaktive Tätigkeiten unterstützen	153
9.1	VBA-Entwicklung in Outlook	154
9.2	Elemente des Outlook-Objektmodells II	154
9.3	Demo-Anwendung „Dateianhänge-Auswahl v1“	156
9.4	Elemente des Outlook-Objektmodells III – Der Outlook-Speicher	162
9.5	Demo-Anwendung „Dateianhänge-Auswahl v2“	165
9.6	Demo-Anwendung „Dateianhänge-Auswahl v3“	170
9.7	Fazit	179
	Literatur	179
10	Web-Dienste und REST APIs	181
10.1	Nice to know: Web Services und REST APIs	182

10.2	Code-Beispiel „Postleitzahlen-API“ und JSON auswerten	189
10.3	Tipps zum Umgang mit JSON in VBA	194
10.4	Code-Beispiel „Spritpreise-Umkreissuche“ mit API-Key	195
10.5	Code-Beispiel „Stimmungserkennung“ mit POST Request	196
10.6	Fazit und Ausblick	197
	Literatur	198
11	SAP GUI Scripting	201
11.1	Objektmodell der SAP GUI	202
11.2	Vorbereitungen für SAP GUI-Scripting	207
11.3	SAP GUI-Skripte ausführen	209
11.4	Demo-Anwendung „Datenpflege“	213
11.5	Tipps für das weitere Vorgehen	223
11.6	Fazit und Ausblick	224
	Literatur	224
12	SOLIDWORKS	225
12.1	VBA in SOLIDWORKS	226
12.2	Das Objektmodell von SOLIDWORKS	227
12.3	Der Objekttyp ModelDoc2	229
12.4	Demo-Anwendung „Volumenkörper“	232
12.5	Fazit	239
	Literatur	239
13	VBA-Tools unternehmenstauglich gestalten	241
13.1	Nice to Know: End User Computing in Forschung und Bildung	242
13.2	Nice to Know: End User Computing im Unternehmen – Chancen und Risiken	243
13.3	Gestaltungstipps für VBA-Tools	245
13.4	Wartungsfreundliche Programmierung	247
13.5	Makros automatisch starten	249
13.6	Ein Makro aus der Symbolleiste für den Schnellzugriff starten	251
13.7	Fazit und Ausblick	253
	Literatur	254

Abbildungsverzeichnis

Abb. 2.1	Aktivieren der Entwicklertools in den Excel-Optionen	9
Abb. 2.2	Die VBA-Entwicklungsumgebung	10
Abb. 2.3	Zuweisen eines Makros an eine Schaltfläche	14
Abb. 2.4	Dialogfenster Extras > Optionen der VBA-Entwicklungsumgebung	18
Abb. 2.5	Mit MsgBox ein Dialogfenster anzeigen	27
Abb. 2.6	Prozeduren in VBA	33
Abb. 2.7	Illustration „Zylinder“ für das Code-Beispiel zu Funktionen	36
Abb. 3.1	Ausschnitt aus dem Objektmodell von Excel	48
Abb. 3.2	Excel-Arbeitsblätter der Demo-Anwendung „Verlinkte Datenblätter“	52
Abb. 3.3	Durch die Demo-Anwendung „Verlinkte Datenblätter“ erzeugte Arbeitsblätter	53
Abb. 3.4	Der Code der Demo-Anwendung „Verlinkte Datenblätter“ in der Entwicklungsumgebung	53
Abb. 4.1	Beispiel Syntaxfehler	63
Abb. 4.2	Beispiel Kompilierungsfehler	63
Abb. 4.3	Beispiel Laufzeitfehler	64
Abb. 4.4	Beispieldaten zur Fehlerbehandlung, links vor und rechts nach Ausführung der Prozedur	69
Abb. 4.5	Objektbibliotheken einbinden	71
Abb. 4.6	Verfügbare und eingebundene Objektbibliotheken	72
Abb. 4.7	Beispielausgabe zu „Referenzierte Objektbibliotheken auflisten“	77
Abb. 5.1	Ausschnitt aus dem Objektmodell der Microsoft Scripting Runtime	80
Abb. 5.2	Einbinden der Objektbibliothek Microsoft Scripting Runtime	81
Abb. 6.1	Ausschnitt aus dem Word-Objektmodell	91
Abb. 6.2	Cursor-Positionen bei der Definition von Range-Objekten	93
Abb. 6.3	Anwendungsbeispiel „Dokumente transformieren zwischen Excel und Word“ – Word-Version	97

Abb. 6.4	Anwendungsbeispiel „Dokumente transformieren zwischen Excel und Word“ – Excel-Version	97
Abb. 6.5	Datenblatt-Vorlage für die Demo-Anwendung „Produktblätter“.....	104
Abb. 6.6	Dokumenteneigenschaften	110
Abb. 7.1	PowerPoint-Vorlage für Code-Beispiele und Demo-Anwendung	119
Abb. 7.2	PowerPoint-Vorlage nach Änderungen durch VBA-Befehle	122
Abb. 7.3	Verweise für die PowerPoint-Demo-Anwendung	123
Abb. 7.4	Muster-PowerPoint-Präsentation für die Demo-Anwendung „PowerPoint-Generator“.....	124
Abb. 7.5	Excel-Arbeitsblatt „Eingabe“ der Demo-Anwendung „PowerPoint-Generator“.....	124
Abb. 7.6	Excel-Arbeitsblatt „Diagrammvorlage“ der Demo-Anwendung „PowerPoint-Generator“.....	125
Abb. 7.7	Excel-Arbeitsblatt „Einstellungen“ der Demo-Anwendung „PowerPoint-Generator“.....	125
Abb. 8.1	Einbinden der Outlook-Objektbibliothek.....	135
Abb. 8.2	Definition von DASL-Filtern	143
Abb. 8.3	Word-Dokument als Vorlage für die Demo-Anwendung „Mailing-Tool“	149
Abb. 8.4	Excel-Arbeitsblatt der Demo-Anwendung „Mailing-Tool“.....	150
Abb. 9.1	UserForm AnhaengeAuswahlForm für die Auswahl von Dateianhängen in Outlook	156
Abb. 9.2	Elemente der AnhaengeAuswahl1-UserForm in Outlook	157
Abb. 9.3	Code der AnhaengeAuswahl1Form-UserForm in Outlook	158
Abb. 9.4	Aufruf des Makros AnhangAuswählen im Menüband des Nachricht -Dialogs	161
Abb. 9.5	Einbinden eines Makros in ein Menüband (Hauptregisterkarte Neue E-Mail-Nachricht des Nachrichten-Inspector)	161
Abb. 9.6	Einbinden eines Makros in die Registerkarte Verfassentools des Outlook-Explorer	162
Abb. 9.7	UserForm zur Eingabe des Dateipfads für die Demo-Anwendung „Dateianhänge-Auswahl v2“	165
Abb. 9.8	UserForm AnhaengeAuswahlForm der Demo-Anwendung „Dateianhänge-Auswahl v2“	168
Abb. 9.9	Verweise auf externe Objektbibliotheken für die Demo-Anwendung „Dateianhänge-Auswahl v3“	171
Abb. 9.10	Aufbau der Demo-Anwendung „Dateianhänge-Auswahl v3“ und vordefinierte Auswahl-UserForm	172
Abb. 10.1	Client und Server	183
Abb. 10.2	REST API	188
Abb. 10.3	JSON-Ausgabe eines GET Requests im Browser	190

Abb. 10.4	Objektbibliotheken für Web Clients	191
Abb. 10.5	Codierte Umlaute in einer HTTP Response	194
Abb. 10.6	Parameter der API Umkreissuche von Tankerkönig.	195
Abb. 11.1	Ausschnitt aus dem Objektmodell der SAP GUI	203
Abb. 11.2	Beispiel für ein SAP GUI-Fenster: MM02	204
Abb. 11.3	Aufbau eines SAP GUI-Fensters, angezeigt im Scripting Tracker [5]	206
Abb. 11.4	Aktivierung und Einstellungsmöglichkeiten für SAP GUI Scripting	208
Abb. 11.5	Der Makrorecorder der SAP GUI	209
Abb. 11.6	Einbinden der SAP GUI Scripting API	214
Abb. 11.7	Beispieldaten zur Demo-Anwendung „Datenpflege“	215
Abb. 11.8	Fehler bei der SAP-Datenpflege: falsches Material	220
Abb. 11.9	Fehler bei der SAP-Datenpflege: falsche Organisationseinheit	221
Abb. 11.10	Fehler beim Speichern bei der SAP-Datenpflege	222
Abb. 12.1	Ausschnitt aus dem SOLIDWORKS Objektmodell, adaptiert und erweitert aus [7, 8]	227
Abb. 12.2	Mit VBA in SOLIDWORKS erzeugte Skizze	233
Abb. 12.3	Mit VBA in SOLIDWORKS erzeugter Volumenkörper	233
Abb. 12.4	Verweise auf die SOLIDWORKS Bibliotheken in Excel-VBA	238
Abb. 13.1	Module im Eigenschaften-Fenster benennen und anordnen	248
Abb. 13.2	Ereignisprozeduren des Word-Objekts Document	250
Abb. 13.3	Einfügen eines Makros in die Symbolleiste für den Schnellzugriff	252
Abb. 13.4	Aufruf eines Makros aus der Symbolleiste für den Schnellzugriff	253

Tabellenverzeichnis

Tab. 2.1	Gebräuchliche elementare Datentypen in VBA	17
Tab. 2.2	Arithmetische Operatoren	21
Tab. 2.3	Operatoren für String-Verkettung.....	22
Tab. 2.4	Operatoren für Vergleiche.....	23
Tab. 2.5	Boolesche Operatoren in VBA.....	24
Tab. 6.1	Mit der Methode Document.Range erzeugte Range-Objekte	93
Tab. 6.2	Beispiel für die Änderung eines Range beim Einfügen und Löschen von Text	95
Tab. 8.1	Outlook-Konstanten für Standard-Outlook-Ordner [2]	134
Tab. 8.2	Outlook-Konstanten für Element-Objekttypen [3]......	138
Tab. 8.3	Intervallkürzel für DateAdd [5]	144
Tab. 10.1	Die häufigsten HTTP Statuscodes [1]	185
Tab. 10.2	Beispiele für Requests an api.zippopotam.us.....	189
Tab. 12.1	Dokumenttypen im SOLIDWORKS-Objektmodell [9]......	230



Einleitung

1

Digitalisierung ist in vielen Bereichen des privaten und öffentlichen Lebens angekommen. Für Unternehmen ist Digitalisierung wichtig, um wettbewerbsfähig zu bleiben und die Anforderungen des Marktes und der Kunden zu erfüllen. Auch Mitarbeiter stellen wachsende Ansprüche an die digitalen Werkzeuge an ihrem Arbeitsplatz. Wirksame digitale Tools reduzieren die Arbeitsbelastung, machen die Arbeit produktiver und letztendlich zeigen sie auch, dass die Leistung der Mitarbeiter wertgeschätzt wird. Die Unternehmen treiben Digitalisierung voran, doch oft bremsen Kapazitätsengpässe in den IT-Abteilungen den Fortschritt. Eine Lösung stellt hier das End User Computing dar: Die User selbst entwickeln Anwendungen und Tools, um sie später für ihre Arbeit zu nutzen. Visual Basic for Applications (VBA) ist seit Langem bekannt und bewährt als Werkzeug, mit dem Endnutzer Lösungen entwickeln, um ihre eigene Arbeit und die Arbeit der Kollegen in den Fachabteilungen komfortabler und effizienter machen. Damit befasst sich „VBA für Office-Automatisierung und Digitalisierung“.

VBA ist eine vollwertige Programmiersprache, die meist innerhalb einer bestehenden Anwendungssoftware, der sogenannten Wirtsanwendung, verwendet wird. Daraus leiten sich viele ihrer Stärken ab. Mit VBA entwickelte Lösungen bauen auf die Wirtsanwendungen auf und können mit deren Funktionen, Bedienoberflächen und Datenspeichermechanismen arbeiten. Dies spart, im Vergleich zu völlig neu entwickelten Lösungen, sehr viel Entwicklungsaufwand. Der Einstieg in die VBA-Entwicklung ist unAufwendig, denn mit den Wirtsanwendungen ist auch VBA bei Entwicklern und Nutzern von VBA-basierten Lösungen bereits installiert und ohne zusätzliche Kosten direkt verwendbar. Da die Nutzer von VBA-Lösungen mit der Wirssoftware vertraut sind, lernen sie meist schnell, mit solchen Tools umzugehen.

Zudem macht VBA es leicht, mehrere VBA-fähige Anwendungssysteme zusammenarbeiten zu lassen. Dadurch erschließen sich weitreichende Möglichkeiten, denn neben der Office-Software von Microsoft sind auch viele Softwarepakete anderer Hersteller mit

VBA ausgestattet. VBA bietet damit das Potenzial, effektive Digitalisierungslösungen mit geringem Aufwand zu realisieren. Diese können auch größere, von der Unternehmens-IT gesteuerte Digitalisierungsprojekte als Prototypen vorbereiten, sie ergänzen oder die Zeit bis zu ihrer Fertigstellung überbrücken.

Besonders häufig wird VBA im Unternehmenskontext zusammen mit Microsoft Excel eingesetzt. Demzufolge sind viele Beispiele und Informationsquellen zu Excel-VBA zu finden. „VBA für Office-Automatisierung und Digitalisierung“ zeigt dagegen Methoden und gibt Anregungen für Digitalisierungslösungen mit VBA über Excel hinaus. Ein Schwerpunkt liegt auf der Software-Integration, also der gleichzeitigen Verwendung mehrerer Softwarepakete in einer VBA-Lösung. Ein eigenes Kapitel befasst sich damit, wie VBA-Lösungen auch Cloud-basierte Dienste nutzen können.

Auch wenn sie durch eine gemeinsame Sprache VBA angesteuert werden, bringen die verschiedenen Anwendungssysteme doch meist eine eigene innere Logik mit, die jeweils ein etwas anderes Herangehen beim Programmieren erfordert. „VBA für Office-Automatisierung und Digitalisierung“ führt in das Objektmodell des jeweiligen Softwaresystems ein und macht seine innere Logik verständlich. Es erläutert die VBA-Entwicklung an kurzen Code-Beispielen und an etwas umfangreicheren Demo-Anwendungen. Da End User Computing nicht nur Vorteile hat, sondern auch Nachteile und Risiken mit sich bringen kann, nennt „VBA für Office-Automatisierung und Digitalisierung“ Maßnahmen, um diese zu minimieren, und thematisiert auch Aspekte der IT-Sicherheit.

Einleitende Kapitel führen in Visual Basic for Applications und in objektbasiertes Programmieren ein und geben einen Einblick in die Welt der Programmiersprachen. Auch in den folgenden Kapiteln sind immer wieder Infos und Tipps zu bewährten Programmierpraktiken zu finden.

Code. Die Code-Beispiele und Demo-Anwendungen sind online in einem Github-Repository verfügbar: <https://github.com/weberi/vba-buch>. Sie sind unter Windows 10 und mit Office 2016 entwickelt und zum großen Teil auch unter Office 365 getestet. Alle Code-Beispiele sind dafür ausgewählt und konzipiert, nützlich und gleichzeitig leicht verständlich zu sein. Dazu gehören auch Kürze und Übersicht. Die Behandlung von Fehlern ist daher oft nur angedeutet. Der Einsatz von Makros, Automatisierungen und IT ganz allgemein bringt Risiken mit sich, die von der jeweiligen konkreten Verwendung und der Umgebung abhängen. Weder Verlag noch Autorin geben irgendeine Zusicherung oder Gewähr, dass die Code-Beispiele wie erwartet funktionieren, sich für einen bestimmten Anwendungszweck eignen oder einen bestimmten konkreten Nutzen stiften. Die Verwendung aller Code-Beispiele, die nicht bereits frei verfügbar sind, wird unter der [Unlicense](#) gestattet, erfolgt aber auf eigenes Risiko.

Anmerkungen zur Sprache. Viele Informationen aus dem Bereich der IT sind auf Englisch geschrieben. Auch die Programmiersprache VBA selbst besteht weitgehend aus englischen Wörtern. Ins Deutsche übersetzte Fachausdrücke lassen sich oft den englischen Originalbegriffen nur schwer wieder zuordnen. Dies führt zu Missverständnissen und erschwert weiterführende Recherchen im Netz. Das Buch verwendet deshalb häufig englische Begriffe im deutschen Text. Es soll auch ohne vertiefte IT-Kenntnisse gut zu

lesen und verständlich sein und verzichtet deshalb möglichst auf Fachjargon. Der Text ist im generischen Maskulin verfasst. Verweise auf Webseiten, zum Beispiel mit Dokumentation oder Downloads, sind unter den Literaturangaben aufgeführt, um lange URLs im Text zu vermeiden.



Einführung in VBA

2

Inhaltsverzeichnis

2.1	Die Programmiersprache VBA	6
2.2	VBA-Programme entwickeln und ausführen.....	8
2.2.1	Die Entwicklungsumgebung	9
2.2.2	VBA-Code schreiben.....	11
2.2.3	VBA-Code in der Entwicklungsumgebung ausführen.....	11
2.2.4	VBA-Code speichern.....	12
2.2.5	VBA-Prozeduren in Anwendungen starten	12
2.2.6	VBA-Code mit einer Schaltfläche starten	13
2.2.7	Der Makrorecorder.....	14
2.3	Variablen, Konstanten und Wertzuweisung	15
2.3.1	Variablen	15
2.3.1.1	Variablennamen, Variablenbezeichner	15
2.3.1.2	Elementare Datentypen in VBA	16
2.3.1.3	Beispiele für Variablendeklarationen	16
2.3.1.4	Der Datentyp Variant	17
2.3.1.5	Mit Option Explicit Variablendeklaration einfordern	18
2.3.2	Wertzuweisungen.....	19
2.3.3	Konstanten	19
2.3.4	Arrays.....	20
2.4	Operatoren	21
2.4.1	Arithmetische Operatoren	21
2.4.2	Verkettungsoperatoren und Stringfunktionen	21
2.4.3	Vergleichsoperatoren	22
2.4.4	Boolesche Operatoren	24
2.5	Schleifen	25
2.5.1	ForNext-Schleifen	25
2.5.2	DoLoop-Schleifen.....	26
2.5.3	ForEach-Schleifen	28

2.5.4 Geschachtelte Schleifen	29
2.6 Verzweigungen	30
2.6.1 IfThenElse-Verzweigung	30
2.6.2 IfThen-Verzweigung	30
2.6.3 IfThenElseIfElse-Verzweigung	31
2.6.4 SelectCase-Verzweigung	31
2.7 Prozeduren und Funktionen	32
2.7.1 Subs definieren und aufrufen	33
2.7.2 Nice to know: Call by Reference versus Call by Value	35
2.7.3 Funktionen definieren und verwenden	35
2.7.4 Optionale und benannte Argumente	37
2.8 Module und Sichtbarkeit von Variablen und Prozeduren	38
2.8.1 Module	39
2.8.2 Sichtbarkeit von Variablen und Konstanten	39
2.8.3 Sichtbarkeit von Prozeduren	41
2.9 Fazit	42

Im Lauf der Jahrzehnte hat die Informatik viele Programmiersprachen entwickelt. Je nach Kontext und Anwendungsbereich sind unterschiedliche Programmiersprachen populär und gebräuchlich. Dabei unterscheiden sich die sprachlichen Grundelemente der verbreiteten Programmiersprachen nur wenig. In der Regel bestimmen andere Faktoren, welche Programmiersprache in einem Projekt zum Einsatz kommt: die Verfügbarkeit geeigneter vorgefertigter Programmbausteine und Code-Bibliotheken, vorhandene Entwicklungswerzeuge und Entwicklungs-Infrastruktur und nicht zuletzt das Know-How und die Vorlieben der Entwickler. Dieses Kapitel ordnet VBA in das Spektrum der Programmiersprachen ein und gibt dann einen Einstieg in die Praxis des Programmierens mit VBA: wie und womit man VBA-Programme erstellt, wie sie aussehen und wie man sie ausführt und speichert. Dies schafft die Voraussetzungen dafür, Code-Beispiele praktisch umzusetzen und auszuprobieren. Der anschließende Überblick über die wichtigsten Sprachelemente von VBA richtet sich an Programmieranfänger wie auch an erfahrene Softwareentwickler, die sonst mit anderen Programmiersprachen arbeiten. Sie schafft die Grundlage dafür, die Code-Beispiele zu verstehen, anzupassen und eigene Programme zu erstellen.

2.1 Die Programmiersprache VBA

VBA ist die Abkürzung von Visual Basic for Applications. BASIC steht für „Beginner’s All-purpose Symbolic Instruction Code“. Es wurde 1964 als besonders leicht zu erlernende Sprache entwickelt mit dem Ziel, sie in Programmierkursen einzusetzen. Wie viele verbreitete Programmiersprachen ist auch BASIC eine imperative Programmiersprache. Ihre Grundbausteine sind Befehle, mit denen Programmierer angeben, was ein Programm tun soll. Nach 1964 wurde BASIC ständig weiterentwickelt. 1991 führte

Microsoft die Programmiersprache Visual Basic ein, mit der man Anwendungen für das Betriebssystem Windows programmieren kann. Im Jahr 1995 begann Microsoft, in seinen Office-Programmen die bis dahin verwendeten Makrosprachen durch Visual Basic for Applications zu ersetzen.

Microsoft Visual Basic for Applications gehört zu den interpretierten Programmiersprachen, im Unterschied zu den kompilierten Programmiersprachen. Interpretierte und kompilierte Programmiersprachen unterscheiden sich darin, wie und wann aus dem Quellcode eines Programms maschinenausführbarer Code wird. Bei einer kompilierten Programmiersprache erzeugt ein Compiler (Übersetzer) eine Datei mit maschinenausführbarem Code, die dann auf Rechnern mit passendem Betriebssystem immer wieder ausgeführt werden kann. Eine interpretierte Programmiersprache benötigt statt eines Compilers einen Interpreter, und zwar bei jeder Ausführung eines Programms. Der Interpreter liest die Befehle als Quellcode ein und führt sie sofort aus.

Interpretierte Programmiersprachen bezeichnet man auch als Skriptsprachen und die damit erstellten Programme als Skripte. Interpretierte Programme laufen langsamer als übersetzte Programme, doch sie sind einfacher zu verbreiten und anzuwenden, weil der Übersetzungsvorgang entfällt und ein Skript überall, wo ein passender Interpreter installiert ist, sofort einsatzfähig ist. Visual Basic for Applications ist in Microsoft Word, Excel, Project, PowerPoint, Access, Visio, Outlook und viele weitere Software-Produkte anderer Hersteller integriert, vor allem in Geschäfts- und Ingenieursanwendungen. Mit diesen Softwaresystemen ist zugleich auch ein Interpreter für VBA auf einem Computer installiert. Mit VBA entwickelt man meist keine eigenständigen Programme, sondern Programmteile, die innerhalb eines dieser Anwendungssoftwaresysteme laufen, es ansteuern und seine Funktionalitäten nutzen. Das umgebende Softwaresystem (die „Application“) bezeichnet man als Wirtsanwendung oder auch Host Application. Neben der jeweiligen Wirtsanwendung lassen sich zugleich auch weitere VBA-fähige Softwaresysteme mit einem Skript ansteuern, sodass diese Systeme nahtlos zusammenarbeiten und Daten austauschen können.

Die technische Basis dafür ist das Component Object Model (COM), das von Microsoft entwickelt und 1992 in das Betriebssystem Windows integriert wurde. COM ist eine objektorientierte Softwarearchitektur, die die Funktionalität von Softwarekomponenten in Objekten kapselt. COM ist plattformunabhängig, was bedeutet, dass die Softwarekomponenten mit beliebigen Programmiersprachen und Entwicklungswerkzeugen entwickelt sein können. Sie werden als Bibliotheken oder als ausführbare Programme bereitgestellt. Die darin implementierten Objekte machen ihre Funktionen über Schnittstellen verfügbar und können von anderen Programmen so aufgerufen und genutzt werden, als seien sie Teil des aufrufenden Programms. Dies ist mit mehreren Programmiersprachen möglich, darunter C++, C#, Delphi, Java, Python und natürlich VBA.

VBA wird oft als *objektbasierte* Programmiersprache bezeichnet. Charakteristisch für *objektorientierte* Programmierung ist, dass sie das entwickelte Softwaresystem durch Klassen und Objekte abbildet, wobei sich die Objekte aus Klassen ableiten. VBA unterstützt es sehr gut, eigene Klassen zu programmieren und daraus Objekte zu erzeugen.

Für einige weiterführende Programmierkonzepte der objektorientierten Programmierung wie Vererbung oder Abstraktion stellt VBA jedoch keine speziellen Sprachmittel zur Verfügung. Während in objektorientierten Programmiersprachen wie Java, C++ oder auch Visual Basic (ohne „Applications“) solche Programmierkonzepte sehr einfach umgesetzt werden können, lassen sie sich in VBA nur umständlich mit anderen Befehlen und Konstruktionen nachbilden. Insofern grenzt sich VBA als objektbasierte Programmiersprache von objektorientierten Programmiersprachen ab. Bei der Entwicklung mit VBA entwickelt man jedoch eher selten eigene Klassen, sondern arbeitet mit den Klassen und Objekten, die die Wirtsanwendungen mitbringen.

Weiterhin unterstützt VBA die ereignisgesteuerte Programmierung. Ereignissteuerung ist wie Objektorientierung ein Programmieransatz, der es ermöglicht, die komplexen Softwaresysteme zu realisieren, die es heute gibt. Statt eine zentrale Ablaufsteuerung zu implementieren, verteilt ereignisgesteuerte Programmierung Zuständigkeiten auf einzelne Softwarekomponenten, in einer objektorientierten Programmiersprache typischerweise auf Objekte. Ereignisse sind dabei zum Beispiel Benutzeraktivitäten wie etwa ein Mausklick auf eine Befehlsschaltfläche oder eine Dateneingabe in ein bestimmtes Eingabefeld, aber auch Nachrichten, die Systemkomponenten untereinander austauschen. Zuständige Softwarekomponenten werden über ein Ereignis informiert und verfügen über Prozeduren und Methoden, um es zu verarbeiten.

VBA ist eine mächtige Programmiersprache mit allen Möglichkeiten, die man von einer modernen, allgemein einsetzbaren Programmiersprache erwarten kann. Sein Sprachkern ist gleichwertig zu dem von C, Python, PhP, C++, Java und so weiter, wobei jede dieser Programmiersprachen ihre speziellen Stärken mitbringt und sich für bestimmte Anwendungszwecke besonders eignet. Für umfangreiche und komplexe Softwareprojekte, bei denen vielleicht sogar mehrere Entwickler mitarbeiten, eignet sich VBA weniger. Die Stärke von VBA ist seine Integration als Makrosprache in funktionsmächtige Anwendungssoftwaresysteme aus dem Office-, Business- und technischen Bereich. VBA eignet sich sehr gut dafür, diese Softwaresysteme zu erschließen, sie zu automatisieren und eigene Tools für spezifische Aufgaben darauf aufzusetzen.

2.2 VBA-Programme entwickeln und ausführen

Ein großes Plus von VBA ist die besonders niedrige Einstiegshürde in die Software-Entwicklung. VBA macht es sehr einfach, Programme zu erstellen und auszuführen. Dazu trägt auch die integrierte Software-Entwicklungsumgebung (IDE) bei, die VBA mitbringt. Im Vergleich zu mächtigen, professionellen Software-Entwicklungswerkzeugen wie Microsoft Visual Studio oder Eclipse hat sie einen geringen Funktionsumfang. Geraade das macht sie aber auch übersichtlich und leicht zu bedienen. Sie lässt sich in Verbindung mit Excel, Word und vielen weiteren VBA-fähigen Anwendungen nutzen. Die jetzt folgenden Erklärungen zur VBA-Entwicklung gelten für Excel als Wirtsanwendung und funktionieren weitgehend genauso mit Word und anderen Office-Wirtsanwendungen.

2.2.1 Die Entwicklungsumgebung

Die VBA-Entwicklungsumgebung lässt sich in Excel mit **Alt-F11** oder aus dem Menüband öffnen. Der Befehl heißt **Visual Basic** und ist auf der Registerkarte **Entwicklertools** zu finden, sofern die **Entwicklertools** aktiviert sind. Falls die Registerkarte **Entwicklertools** nicht im Menüband erscheint, kann man sie sichtbar machen, indem man auf der Registerkarte **Datei** den Menüpunkt **Optionen** anklickt und im sich öffnenden **Optionen**-Dialogfenster den Eintrag **Menüband anpassen** wählt. Dann hakt man unter **Hauptregisterkarten** das Kontrollkästchen **Entwicklertools** an und verlässt das Dialogfenster mit **Ok**, siehe Abb. 2.1.

Abb. 2.2 zeigt eine Ansicht der VBA-Entwicklungsumgebung mit vier Fenstern. Weitere oder andere Fenster können im DropDown-Menü **Ansicht** sichtbar gemacht werden. In Abb. 2.2 ist rechts das große **Code**-Fenster zu sehen. Das Fenster darunter ist der **Direktbereich**. In der linken Spalte sind der **Projekt-Explorer** und darunter das **Eigenschaften**-Fenster angeordnet.

Der **Projekt-Explorer** listet die Komponenten von VBA-Projekten auf. Bei der VBA-Entwicklung beginnt man meist damit, in den **Projekt-Explorer** zu klicken und im Kontextmenü **Einfügen > Modul** zu wählen. Der **Projekt-Explorer** legt dann ein neues Modul an und öffnet es im **Code**-Fenster. Ein solches Modul wird als Code-Modul oder Standard-Modul bezeichnet. Hier kann man VBA-Code eingeben. Das Modul „Modul1“ in Abb. 2.2 enthält bereits ein kleines Skript mit einer Sub `HalloWelt`. Im Feld **(Name)** des **Eigenschaften**-Fensters lässt sich das Modul umbenennen.

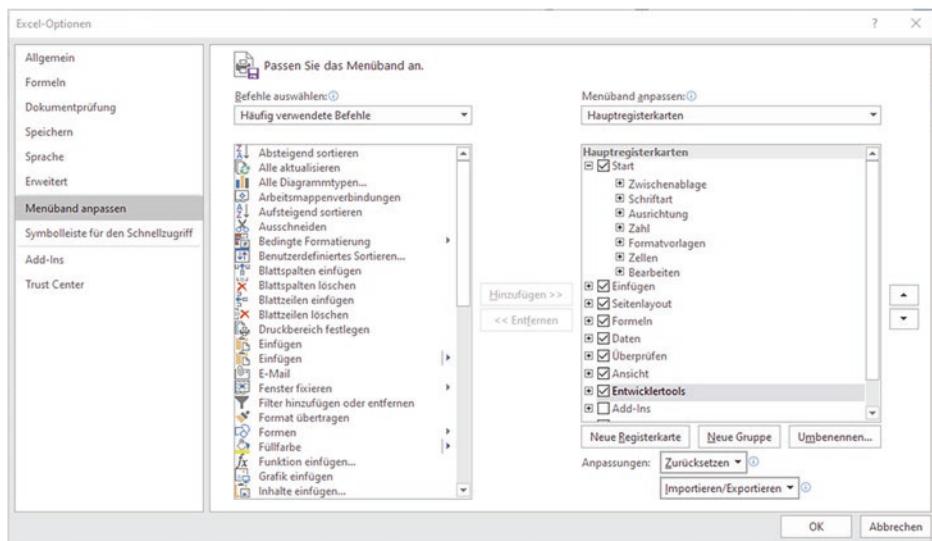


Abb. 2.1 Aktivieren der Entwicklertools in den Excel-Optionen

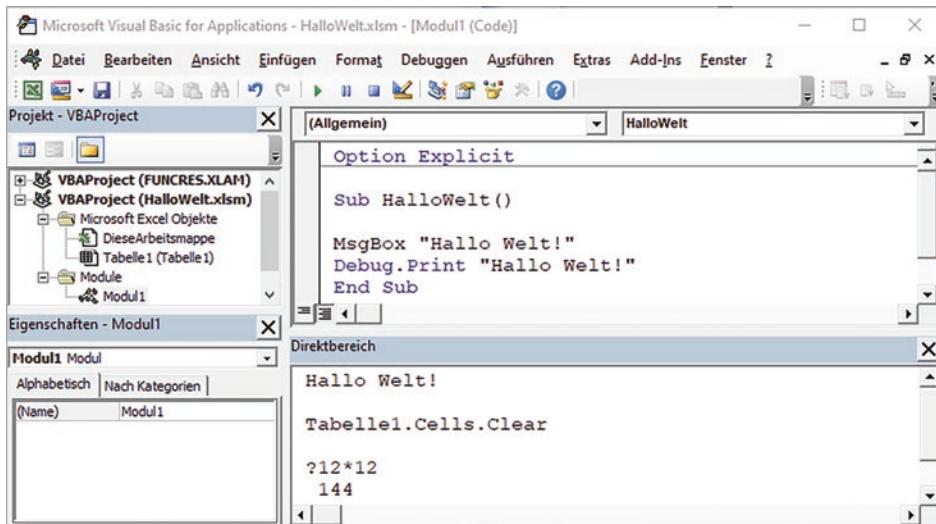


Abb. 2.2 Die VBA-Entwicklungsumgebung

Wenn man eines der **Microsoft Excel Objekte** im **Projekt-Explorer** anklickt (zum Beispiel „Tabelle1“ oder „DieseArbeitsmappe“), öffnet sich ebenfalls ein Modul, ein sogenanntes Objektmodul. Auch ein Objektmodul kann Code aufnehmen. Es ist jedoch nicht für allgemein verwendete Programmteile vorgesehen, sondern für objekt-spezifischen Code. Weitere Infos zu Modulen gibt Abschn. 2.7.4.

Der **Direktbereich** dient als Eingabe- und Ausgabefenster für den VBA-Interpreter. Hier erscheinen Ausgaben von `Debug.Print`-Befehlen. In Abb. 2.2 zeigt der Direktbereich oben die Ausgabe des `Debug.Print`-Befehls aus der Sub `HalloWelt`. Wenn man einen Befehl in den **Direktbereich** schreibt und mit der Eingabetaste abschließt, führt ihn der VBA-Interpreter sofort aus. Hier kann man Befehle ausprobieren, den Interpreter einen Ausdruck auswerten lassen oder Aktionen in Excel ausführen, zum Beispiel schnell Zelleninhalte und Formatierungen von einem Arbeitsblatt löschen, die beim Entwickeln und Testen eines Skripts entstanden sind. Um einen Ausdruck auszuwerten, tippt man ein Fragezeichen und den Ausdruck (zum Beispiel „`12*12`“) in den Direktbereich und schließt mit der Eingabetaste ab.

Nützlich sind auch das **Lokal**-Fenster und der **Objektkatalog**, die beide in Abb. 2.2 nicht sichtbar sind. Sie lassen sich im DropDown-Menü **Ansicht** öffnen. Das **Lokal**-Fenster zeigt Variablen und ihren Wert, während der Interpreter ein Skript ausführt. Der **Objektkatalog**, der auch mit der Funktionstaste **F2** zu öffnen ist, listet die jeweils verfügbaren Objekte auf. Er überlagert das **Code**-Fenster. Die Funktionstaste **F7** bringt das **Code**-Fenster wieder nach oben.

2.2.2 VBA-Code schreiben

VBA ist eine zeilenorientierte Sprache. Während andere Programmiersprachen das Ende eines Befehls zum Beispiel mit einem Strichpunkt „;“ markieren, endet ein VBA-Befehl mit der Zeile, in der er steht. Sehr lange Befehle lassen sich auf mehrere Zeilen umbrechen, indem man eine Zeile mit einem Leerzeichen und einem darauffolgenden Unterstrich beendet und den Befehl auf der folgenden Zeile fortsetzt. Getrennt von einem Doppelpunkt lassen sich mehrere Befehle in eine Zeile schreiben. Man sieht diese Schreibweise gelegentlich bei Variablendeklarationen mit anschließender Initialisierung der Variablen. Sie macht den Code kompakter, aber auch weniger übersichtlich. Der folgende Code-Schnipsel zeigt Beispiele für VBA-Befehle.

```
Sub SyntaxDemo()

Dim i As Integer: i = 1
Dim info As String

info = "ich bin ein ziemlich " _
& "langer Infotext."

' Debug.Print info      ' auskommentiert!
Debug.Print info        ' nicht auskommentiert!
End Sub
```

Kommentare sind Beschreibungen, Erklärungen, Notizen oder auch ungenutzte Code-Teile, die bei der Ausführung des Programms ignoriert werden sollen. Kommentare markiert man in VBA durch ein Hochkomma. Das Hochkomma kann am Anfang einer Zeile stehen oder auch hinter einem Befehl. Alle Zeichen, die in der betreffenden Zeile auf das Hochkomma folgen, sind Kommentar. VBA kennt keine weiteren Möglichkeiten, Text als Kommentar auszuzeichnen. Wenn man mit Kommentarzeichen vorübergehend Code-Teile „stillegt“, nennt man dies auch „Auskommentieren“. Auskommentieren hilft bei der Programmierung, um Fehler einzuschränken, noch nicht vollständig programmierte Code-Teile zu überspringen oder um beim Testweisen Programmausführen langdauernde, unkritische Berechnungen auszulassen.

2.2.3 VBA-Code in der Entwicklungsumgebung ausführen

Alle VBA-Befehle in einem Code-Modul müssen in einer Prozedur stehen, also zwischen Sub und End Sub. Prozeduren wie die beiden Code-Beispiele HalloWelt in

Abb. 2.2 und die eben gezeigte SyntaxDemo kann man in der Entwicklungsumgebung direkt ausführen. Dazu platziert man den Cursor in den Bereich zwischen Sub und End Sub und drückt die Funktionstaste **F5** oder wählt den Befehl **Sub/UserForm ausführen** im Menüband der Entwicklertools oder im DropDownList-Menü **Ausführen**. Für die Fehler-suche oder um nachzuvollziehen, was ein Codestück genau bewirkt, führt man den Code in Einzelschritten aus. Statt der Funktionstaste **F5** drückt man dazu für jeden auszuführenden Befehl die Funktionstaste **F8** oder wählt die Funktion **Einzelschritt** im DropDownList-Menü **Debuggen**.

Direkt ausführbare Prozeduren oder „Subs“ bezeichnet man in VBA-Sprechweise auch als Makro. Sie beginnen mit dem Schlüsselwort Sub und das Klammerpaar () hinter ihrem Bezeichner ist leer.

2.2.4 VBA-Code speichern

Wenn man VBA in einer Office-Wirtsanwendung entwickelt, speichert man den VBA-Code in einer Datei der Wirtsanwendung. Office-Dateien mit Makros sind an ihrer Dateiendung zu erkennen: „.xlsm“ für Excel-Arbeitsmappen mit VBA-Code, „.docm“ für Word-Dokumente mit VBA-Code, „.pptm“ für PowerPoint-Präsentationen mit VBA-Code und so weiter. Das „.m“ in der Dateiendung steht dabei für „Makros“. Eine Office-Datei mit Makros hat ein anderes Format als eine Office-Datei ohne Makros. Beim Speichern der Datei muss man das Dateiformat entsprechend umstellen, sonst geht der VBA-Code verloren.

Wirtsanwendungen anderer Hersteller gehen möglicherweise anders mit VBA-Code um. In jedem Fall besteht ein VBA-Quellprogramm aus ganz gewöhnlichem Text und kann mit allen üblichen Texteditoren bearbeitet und als Textdatei gespeichert werden. VBA-Quellcode lässt sich wie anderer Text auch per Copy&Paste zwischen der Entwicklungsumgebung und anderen Dateien übertragen. Die Entwicklungsumgebung bietet außerdem eine Funktion **Datei exportieren**, die ein VBA-Modul in eine Textdatei exportiert. VBA-Dateien haben üblicherweise die Dateiendung „.bas“. Die Funktion **Datei exportieren** findet man in dem Kontextmenü, das sich bei Rechtsklick auf ein Modul im **Projekt-Explorer** öffnet, vergleiche Abb. 2.2. Auch eine Funktion **Datei importieren** ist dort zu finden.

Weitere Funktionen zum Verwalten von VBA-Code bietet das Dialogfenster **Projekt-Eigenschaften**, das man im DropDownList-Menü **Extras** der VBA-Entwicklungsumgebung öffnen kann. Hier kann man unter anderem einen Namen und eine Beschreibung für das VBA-Projekt festlegen und ein Kennwort angeben, wenn man den VBA-Code verstecken möchte.

2.2.5 VBA-Prozeduren in Anwendungen starten

In echten Anwendungen startet man VBA-Code natürlich nicht aus der Entwicklungsumgebung. VBA bietet vielfältige Möglichkeiten, um VBA-Skripte in einer Anwendung zugänglich zu machen.

- In Kap. 9 wird beschrieben, wie man eine mit VBA realisierte Funktion in das Menüband einer Office-Anwendung einbinden kann.
- Ein Symbol für eine solche Funktion lässt sich auch in die Symbolleiste für den Schnellzugriff integrieren. Wie das geht, erklärt ein Beispiel in Abschn. 13.6.
- Weiterhin kann man ein VBA-Skript automatisch anstoßen, wenn der Benutzer eine bestimmte Aktion ausführt, zum Beispiel beim Öffnen einer Arbeitsmappe. Diesen Ansatz nutzt und demonstriert ein Beispiel in Abschn. 13.5.
- Wenn man in Excel als Wirtsanwendung entwickelt, ist ein Button (Schaltfläche) eine sehr gute Lösung, um ein VBA-Skript zu starten. Dass sie auch sehr schnell und einfach zu realisieren ist, zeigt der folgende Abschnitt.

Auch ein Tastatorkürzel ist schnell als Starter für ein VBA-Skript eingerichtet. Dies empfiehlt sich für VBA-basierte Tools jedoch weniger, weil Benutzer es auf der Bedienoberfläche nicht finden können. Statt irgendwo einen Text zu platzieren, der auf die Funktion und das Tastatorkürzel hinweist, ist ein klickbares Bedienelement wie die bereits genannte Schaltfläche meist bedienfreundlicher.

2.2.6 VBA-Code mit einer Schaltfläche starten

Um ein Skript mit einer Schaltfläche zu verbinden, braucht man zunächst ein Skript. Das folgende kleine Demo-Skript genügt zum Ausprobieren. Man schreibt es wie in Abb. 2.2 in der Entwicklungsumgebung von Excel in ein Code-Modul.

```
Sub HalloWelt()  
    MsgBox "Hallo Welt!"  
End Sub
```

Schaltflächen und weitere Steuerelemente findet man auf der Registerkarte **Entwicklertools** unter dem Werkzeugkoffer-Symbol mit der Beschriftung **Einfügen** in der Befehlsgruppe **Steuerelemente**. Am einfachsten ist die Schaltfläche zu benutzen, die im Bereich **Formularsteuerelemente** oben links angeordnet ist.

Nachdem man auf dem Arbeitsblatt eine solche Schaltfläche aufgezogen hat, öffnet sich sofort das Dialogfenster **Makro zuweisen** und bietet alle ausführbaren Subs an. In Abb. 2.3 gibt es lediglich das Makro „HalloWelt“, das man der Schaltfläche zuweisen kann. „Schaltfläche4_Klicken“ ist ein Vorschlag der Entwicklungsumgebung, um ein Makro zu bezeichnen, das man mit **Neu** oder **Aufzeichnen** erst generieren würde. **Neu** führt dabei in ein Codefenster der Entwicklungsumgebung und **Aufzeichnen** startet den Makrorecorder, der in Abschn. 2.2.7 beschrieben wird. Den Bezeichner „Schaltfläche4_Klicken“ sollte man nur für einmalig verwendeten Wegwerfcode beibehalten. Für dauerhaft genutzte Makros lohnt sich ein sprechenderer Bezeichner.

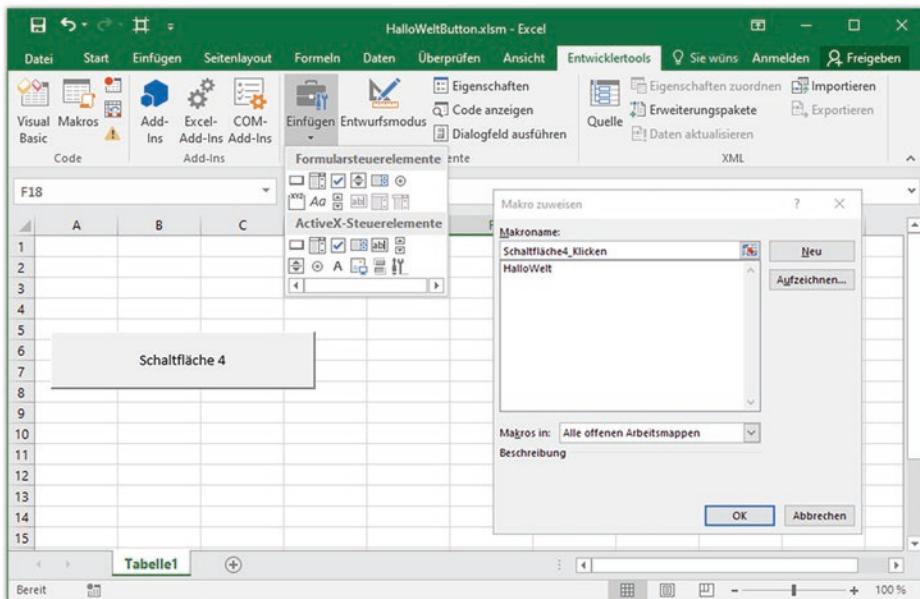


Abb. 2.3 Zuweisen eines Makros an eine Schaltfläche

Ein Rechtsklick auf die neu eingefügte Schaltfläche öffnet ihr Kontextmenü. Hier kann man ihre Beschriftung ändern und sie etwas formatieren. Die entsprechenden Menüpunkte heißen **Text bearbeiten** und **Steuerelement formatieren**. Wem die Formatierungs- und Gestaltungsmöglichkeiten der Schaltfläche nicht ausreichen, kann stattdessen und genauso einfach eine Form aus dem **Formen**-Katalog als Bedienelement einsetzen. Die Formen findet man in der Befehlsgruppe **Illustrationen** der Registerkarte **Einfügen**. Sie lassen sich sehr flexibel designen und haben ebenfalls den Eintrag **Makro zuweisen** im Kontextmenü. Auch ein Bild kann man auf diese Weise mit einem Makro verknüpfen.

2.2.7 Der Makrorecorder

Excel, Word und auch weitere Anwendungssysteme besitzen einen VBA-Makrorecoder, mit dem Benutzer Aktionen aufzeichnen können, während sie sie in der Software ausführen. Der Makrorecoder schreibt die Aktionen als VBA-Code in der Entwicklungs-Umgebung in ein Standard-Modul. Dort kann man den Code ansehen und bearbeiten. Solche aufgezeichneten Makros sind dafür gedacht, häufig wiederkehrende Arbeitsabläufe auf die Schnelle zu automatisieren. Der automatisch generierte Code ist nach einem typischen Muster und eher umständlich aufgebaut und lässt sich oft nicht ohne

Bearbeitung weiterverwenden, doch er ist eine gute Option, um herauszufinden, wie VBA-Befehle heißen und aufgerufen werden.

Nicht alle VBA-fähigen Softwaresysteme bieten einen Makrorecorder. Ob und wo er zu finden ist, hängt von der jeweiligen Anwendung ab. In den meisten Office-Programmen liegen die Befehle zum Starten und Stoppen des Makrorecorders in der Befehlsguppe **Code** auf dem Menüband der **Entwicklertools**. In Excel und Word sind sie auch aus der Statuszeile am unteren Rand des Anwendungsfensters zugänglich, sofern die Entwicklertools aktiviert sind.

2.3 Variablen, Konstanten und Wertzuweisung

Alle Daten, mit denen ein Computerprogramm arbeiten soll, müssen im Arbeitsspeicher des Computers abgelegt werden. Eine Variable ist ein Speicherplatz für ein Daten-element. Bei der Programmierung „deklariert man eine Variable“, um dem System mit-zuteilen, dass das entwickelte Programm einen Speicherplatz für ein Datenelement be-nötigt. Dieser Abschnitt erklärt, wie in VBA Variablen und andere Speicherstrukturen für einfache Datenelemente deklariert und mit Werten befüllt werden.

2.3.1 Variablen

Bei der Deklaration einer Variablen legt man einen Bezeichner für die Variable fest, um später auf sie zugreifen zu können. Zudem gibt man an, welche Art von Datenwert die Variable aufnehmen soll, den sogenannten Datentyp. Damit kann das System Speicher in der passenden Größe für die Variable reservieren und auch verhindern, dass ein fehler-hafes Programm unzulässige Operationen mit den in der Variable gespeicherten Werten durchführt. In VBA ist die Angabe eines Datentyps optional, aber sehr nützlich. Eine Va-riable hat außer einem Namen und einem Typ auch noch einen Gültigkeitsbereich. Die-ses Thema wird später noch behandelt.

2.3.1.1 Variablennamen, Variablenbezeichner

Der Name einer Variablen ist fast frei wählbar. Er soll den Verwendungszweck der Va-riablen und der darin gespeicherten Werte ausdrücken. Für Variablennamen gelten fol-gende formale Einschränkungen:

- Ein Variablenname muss mit einem Buchstaben beginnen. Er kann Ziffern enthalten, aber nicht als erstes Zeichen.
- Ein Variablenname kann höchstens 250 Zeichen lang sein.
- Ein Variablenname ist eine zusammenhängende Zeichenkette, enthält also keine Leer-zeichen.

- Zum Gliedern langer Variablennamen verwendet man Unter_Striche oder die sogenannte „CamelCase“- oder HöckerSchreibWeise, korrekt Binnen-Majuskeln. Bindestriche funktionieren nicht, weil der VBA-Interpreter sie als Minus-Zeichen versteht.
- VBA unterscheidet nicht zwischen groß oder klein geschriebenen Buchstaben in Variablenbezeichnern. Groß- oder Kleinschreibung von Buchstaben in Variablenbezeichnern dient also nur der Optik.

Manche Visual Basic-Programmierer richten sich nach einer Konvention, bei der der Datentyp einer Variablen in ihren Bezeichner codiert wird. Zum Beispiel beginnen dann die Bezeichner aller Stringvariablen mit `str`, also etwa `strNachname`. In neuerer Zeit hat diese Konvention an Bedeutung verloren.

2.3.1.2 Elementare Datentypen in VBA

Die wichtigsten elementaren Datentypen der Programmiersprache VBA sind in Tab. 2.1 zusammengestellt.

Den Datentyp `String` verwendet man für Zeichenketten, die aus Buchstaben, Ziffern, Satz- und anderen Sonderzeichen bestehen und auch sogenannte Whitespace-Zeichen wie Leerzeichen und Tabulatoren enthalten können. Der Datentyp `String` kann mit einer festen Länge oder mit einer dynamischen Länge deklariert werden. Deklariert man Strings mit fester Länge, belegt das Programm weniger Speicher und läuft etwas schneller. Mit Strings variabler Länge ist es flexibler. Strings mit fester Länge verwendet man nur selten.

Auch die numerischen Datentypen benötigen unterschiedlich viel Speicher. Da moderne Rechner in der Regel ausreichend Arbeitsspeicher besitzen, verwendet man für Gleitkommazahlen in der Regel immer den Datentyp `Double`. Der Ganzzahlen-Datentyp `Integer` kann nur kleine Zahlen aufnehmen. Er reicht zum Beispiel nicht annähernd, um die 1 Million Zeilen eines Excel-Arbeitsblatts zu indizieren. Man verwendet ihn, um zum Beispiel eine Liste oder eine Matrix mit nur wenigen Elementen zu durchlaufen. Sonst eignet sich der Datentyp `Long` besser.

2.3.1.3 Beispiele für Variablen-deklarationen

In VBA wird eine Variablen-deklaration mit dem Schlüsselwort `Dim` eingeleitet. Der folgende Code-Schnipsel zeigt einige Beispiele.

```
Dim endeErreicht As Boolean
Dim zeile As Long
Dim gewicht As Double
Dim start As Date
Dim erfolgsmeldung As String
Dim diverses As Variant
' Dim diverses           ' kein expliziter Typ -> Variant, vermeiden!!
```

Tab. 2.1 Gebräuchliche elementare Datentypen in VBA

Typ	Verwendung	Reservierter Speicherplatz
Boolean	Kann nur einen der beiden Werte Wahr oder Falsch annehmen	2 Byte
Integer	Ganze Zahlen von -32.768 bis +32.767	2 Byte
Long	Ganze Zahlen von -2.147.483.648 bis +2.147.483.647	4 Byte
LongLong	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807	8 Byte, nur auf 64-Bit-Plattformen verfügbar
Currency	Festkommazahl mit 15 Vor- und 4 Nachkommastellen von -922.337.203.685.477,5808 bis 922.337.203.685.477,5807	8 Byte
Single	Gleitkommazahl zwischen $\pm 3,402823 \times 10^{38}$ und $\pm 1,401298 \times 10^{-45}$ mit einer Präzision von 6 Stellen nach dem Komma, und die Zahl 0	4 Byte
Double	Gleitkommazahl im Bereich von $-1,79769313486231 \times 10^{308}$ bis $-4,94065645841247 \times 10^{-324}$ für negative Werte und $4,94065645841247 \times 10^{-324}$ bis $1,79769313486232 \times 10^{308}$ für positive Werte	8 Byte
Date	Zeitpunkt zwischen 01. Januar 100 und 31. Dezember 9999 mit Uhrzeit von 0:00:00 bis 23:59:59	8 Byte
String *n	String mit fest vorgegebener Länge n im Bereich von 0 bis etwa 65.400 Zeichen	n Byte
String	String mit variabler Länge von 0 bis ungefähr 2 Mrd. Zeichen	10 Byte + Länge der Zeichenfolge

Den Befehl `Dim zeile As Long` kann man lesen als „Dimensioniere eine Variable `zeile` so, dass ein `Long`-Wert hineinpasst“. Den Befehl `Dim start As Date` kann man lesen als „Dimensioniere eine Variable `start` so, dass ein Zeitpunkt hineinpasst“.

2.3.1.4 Der Datentyp Variant

Ein weiterer, spezieller Datentyp in VBA ist `Variant`. Eine Variable dieses Typs kann alle elementaren Datenarten aus Tab. 2.1 speichern. In VBA ist die Angabe von Datentypen für Variablen optional. Man kann sie einfach weglassen und die Variable erhält dann implizit den Datentyp `Variant`. Mit expliziten und genau passenden Datentyp-Deklarationen ist ein Programm aber nicht nur etwas schneller und speichereffizienter. Viel wichtiger ist dabei, dass eine Typ-Deklaration auch die beabsichtigte Verwendung

der Variablen verdeutlicht und damit den Code stärker selbsterklärend und Programmierfehler erkennbar macht. Wo immer möglich, deklariert man daher Variablen mit dem passenden spezifischen Datentyp aus Tab. 2.1. Es gibt jedoch in VBA einige spezielle Programmkonstrukte, die den Datentyp Variant verlangen. Es ist dann gute Programmierpraxis, die betreffenden Variablen explizit als Variant zu deklarieren, um zu verdeutlichen, dass dieser Datentyp hier beabsichtigt ist.

2.3.1.5 Mit Option Explicit Variablen-deklaration einfordern

VBA ist keine strenge Sprache und macht das Programmieren leicht. Man muss Variablen nicht deklarieren, sondern kann sie ohne Deklaration einfach benutzen. Wenn ein unbekanntes Wort im Code auftaucht, betrachtet es der Interpreter als eine neue Variable. Die Anweisung `Option Explicit` in einem Modul bewirkt, dass der Interpreter undeclared Variablen nicht mehr zulässt und eine Fehlermeldung erzeugt, wenn er im Code ein unbekanntes, nicht deklariertes Wort entdeckt. Dadurch werden viele Programmierfehler und Vertipper sofort sichtbar.

Die Anweisung `Option Explicit` muss am Anfang eines Moduls stehen. Man kann sie dort selbst einfach eintragen. Einfacher und sicherer ist es jedoch, wenn die Entwicklungsumgebung sie automatisch in jedes neue Modul einfügt. Dazu wählt man im DropDownList-Menü **Extras** den Menüpunkt **Optionen** und setzt einen Haken bei „Variablen-deklaration erforderlich“, siehe Abb. 2.4.

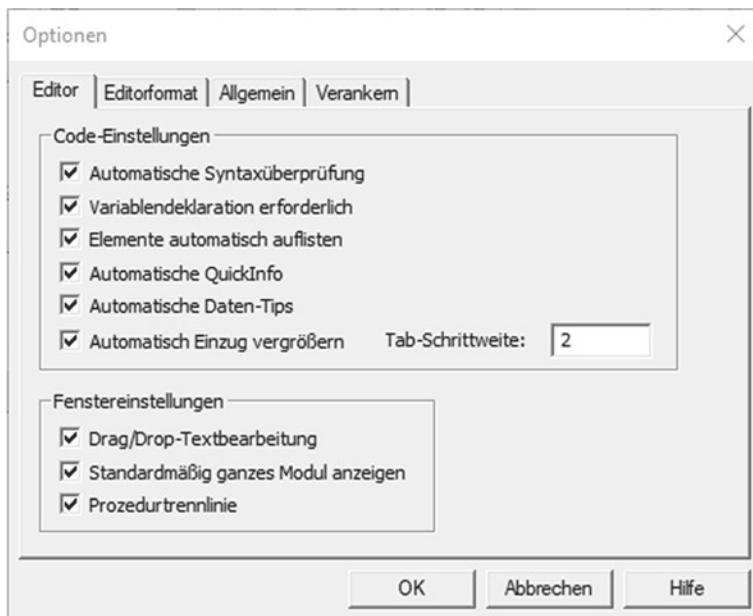


Abb. 2.4 Dialogfenster **Extras > Optionen** der VBA-Entwicklungsumgebung

2.3.2 Wertzuweisungen

Wie kann man nun einen bestimmten Datenwert in eine Variable speichern? Dafür gibt es den Zuweisungsoperator. In Visual Basic wird er durch das Zeichen = dargestellt. Achtung, dieses = ist ganz anders als das mathematische Gleichheitszeichen! Es stellt keine Aussage über Gleichheit dar, sondern steht für eine Operation, also einen Vorgang. Es ist nicht symmetrisch, sondern hat eine Richtung: es weist der Variable links vom Gleichheitszeichen den Wert zu, der rechts vom Gleichheitszeichen steht. Links vom Gleichheitszeichen muss eine Variable stehen. Rechts kann ein Wert, eine Funktion, ein Ausdruck oder ebenfalls eine Variable stehen. Im letzten Fall erhält die linke Variable den Wert zugewiesen, den die rechte Variable speichert.

Dieses Code-Beispiel zeigt einige Wertzuweisungen:

```
Sub VariablenDemo()
    Dim a As Long      ' Variablen Deklarationen
    Dim b As Long
    Dim c As Long
    Dim d As Long
    Dim erg as String
    a = 50            ' Wertzuweisung
    b = 5
    c = b            ' c erhält den Wert von b
    c = c + 1        ' der Wert von c wird um 1 erhöht
    d = a + b        ' Addition mit 2 Variablen und Wertzuweisung

    erg = "c = " & c & " und d = " & d ' baut einen String und speichert
                                         ' ihn in eine Variable
    Debug.Print erg
End Sub
```

2.3.3 Konstanten

Variablen dienen als Speicherplatz für wechselnde Werte. Konstanten speichern Werte, die sich während des Programmablaufs nicht mehr ändern. Eine Konstante deklariert man ähnlich wie eine Variable, weist ihr aber sofort einen Wert zu, den man später nicht mehr ändern kann.

```
Const erfolgsmeldung As String = "Vorgang erfolgreich ausgeführt!"
Const gravitation As Single = 6.67384E-11
```

Konstanten, die Geltung für ein ganzes VBA-Projekt haben und in mehreren Programmteilen verwendet werden, bezeichnet man auch als globale Konstanten. Es ist oft nützlich, sie optisch hervorzuheben. Man schreibt dazu den Konstantenbezeichner komplett in Großbuchstaben wie hier:

```
Const MAX_TEILNEHMER As Integer = 5
```

2.3.4 Arrays

Ein Array ist ein Feld gleichartiger Datentypen. Man verwendet ein Array, wenn man mehrere, ähnlich verwendete Variablen mit dem gleichen Datentyp braucht, oder um Vektoren oder Matrizen zu speichern.

Ein Array (Feld) wird deklariert wie eine einzelne Variable mit dem Unterschied, dass angegeben wird, wie viele Speicherplätze das Array umfassen soll. Auf einen einzelnen Wert in einem Array greift man über seinen Index zu. Die zulässigen Indizes sind in der Variablendeclaration im Ausdruck `n To m` festgelegt. Versucht man, auf einen Wert außerhalb des Arrays zuzugreifen, reagiert das System mit einer Fehlermeldung: „Index außerhalb des gültigen Bereichs“.

```
Sub ArrayDemo1()
Dim vektor(1 To 5) As Integer

vektor(1) = 100
vektor(2) = 150
vektor(3) = 170
vektor(4) = 200
vektor(5) = 201
Debug.Print vektor(2) + vektor(3)           ' 320
vektor(0) = 3 ' Index-Fehler!!
End Sub
```

Man kann ein- oder mehrdimensionale Arrays deklarieren. Für jede Dimension deklariert man den zulässigen Wertebereich des Index in der Form `n To m` und listet die Angaben durch Komma getrennt auf. Bis zu 60 Dimensionen sind möglich.

```
Sub ArrayDemo2()
Dim matrix(0 To 1, 0 To 2) As Double

matrix(0, 0) = 13.05
matrix(0, 1) = 1.002
matrix(0, 2) = 33
matrix(1, 0) = 65.88
matrix(1, 1) = 43.6
```

```
matrix(1, 2) = 32.23
```

```
Debug.Print matrix(1, 1)
End Sub
```

2.4 Operatoren

Als Operator bezeichnet man Zeichen oder Zeichenkombinationen, die einen Rechenvorgang oder eine andere Operation auf Datenwerten ausdrücken. Für die verschiedenen Datentypen gibt es jeweils passende Operatoren. Einige Operatoren wurden bereits in Code-Beispielen verwendet, darunter der Zuweisungsoperator =.

2.4.1 Arithmetische Operatoren

Eine Übersicht über arithmetische Operatoren in VBA zeigt Tab. 2.2.

2.4.2 Verkettungsoperatoren und Stringfunktionen

Auch der Verkettungsoperator & war bereits im Code-Beispiel zu sehen. Der Fachausdruck für String-Verkettung lautet „Konkatenation“. VBA kennt zwei Operatoren für String-Verkettung, siehe Tab. 2.3.

Tab. 2.2 Arithmetische Operatoren

Zeichen	Funktion	Anwendbar für
+	Addition	Alle numerischen Typen
-	Subtraktion	Alle numerischen Typen
*	Multiplikation	Alle numerischen Typen
/	Division	Alle numerischen Typen, Ergebnis ist ein Double (Ausnahmen siehe Dokumentation)
\	Ganzzahlige Division, Rest wird ignoriert	Alle numerischen Typen (Gleitkommawerte werden gerundet), Ergebnis ist ein Ganzzahltyp (Integer, Long)
Mod	Modulo, Rest bei der ganzzahligen Division	Alle numerischen Typen (Gleitkommawerte werden gerundet), Ergebnis ist ein Ganzzahltyp (Integer, Long)
^	Potenz, Zahl^Exponent	Alle numerischen Typen, Ergebnis ist ein Double

Tab. 2.3 Operatoren für String-Verkettung

Zeichen	Funktion	Anwendbar für
&	Verkettet zwei Ausdrücke zu einem String	Beliebige Ausdrücke. Nicht-String-Typen werden in einen String umgewandelt
+	Verkettet zwei Strings zu einem String	Anwendbar auf zwei Strings. Angewendet auf einen String und eine Zahl: Fehler!

```
Sub KonkatDemo()
Dim konkat As String
Dim anfang As String
Dim ende As String
anfang = "Anfang "
ende = " und Ende"
konkat = anfang & ende
Debug.Print konkat
End Sub
```

VBA bietet einige Funktionen, um mit Strings umzugehen. Der folgende Demo-Code zeigt Beispiele.

```
Sub StringFunktionen()
Dim text1 As String
text1 = "Otto Müller, Kempten"

Debug.Print text1
Debug.Print Len(text1)           ' Länge des Strings
Debug.Print InStr(text1, "te")    ' Startposition eines Teilstrings im
                                 ' String
Debug.Print Mid(text1, 5, 6)      ' Teilstring ab Position 5, 6 Zeichen
                                 ' lang
Debug.Print Left(text1, 4)        ' Linker Teilstring, 4 Zeichen lang
Debug.Print Right(text1, 8)       ' Rechter Teilstring, 8 Zeichen lang
Debug.Print StrReverse(text1)    ' String umgedreht
Debug.Print Replace(text1, "t", "!!")  ' Zeichen t durch !! ersetzt
End Sub
```

2.4.3 Vergleichsoperatoren

Das Ergebnis einer Vergleichsoperation ist ein Boolescher Wert, also entweder True (Wahr) oder False (Falsch). Tab. 2.4 zeigt eine Übersicht.

Tab. 2.4 Operatoren für Vergleiche

Zeichen	Funktion	Anwendbar für
<	Kleiner	Beliebige Ausdrücke
<=	Kleiner oder gleich	Beliebige Ausdrücke
>	Größer	Beliebige Ausdrücke
>=	Größer oder gleich	Beliebige Ausdrücke
=	Gleich	Beliebige Ausdrücke; ob = eine Zuweisung oder einen Vergleich darstellt, hängt davon ab, wo im Code es steht
<>	Ungleich	Beliebige Ausdrücke

Zwei numerische Werte werden als Zahlen verglichen. Zwei Strings werden lexikalisch (alphabetisch) verglichen. Beim lexikalischen Vergleich sind Ziffern (Zahlenzeichen) kleiner als Buchstaben.

Wenn ein String mit einem numerischen Wert verglichen wird und sich der String in eine Zahl umwandeln lässt, vergleicht VBA die Werte numerisch, also als Zahlen. Wenn ein String mit einem numerischen Wert verglichen wird und sich der String nicht in eine Zahl umwandeln lässt, gibt es eine Fehlermeldung.

```
Sub VergleicheDemo()
Dim i1 As Integer: i1 = 10
Dim i2 As Integer: i2 = 2
Dim s1 As String: s1 = "10"
Dim s2 As String: s2 = "2"
Dim s3 As String: s3 = "eins"
Dim erg As Boolean

Debug.Print "zwei Zahlen vergleichen:"
erg = i1 < i2                      ' Falsch
Debug.Print i1 <> i2                ' Wahr

Debug.Print "zwei Strings vergleichen:"
Debug.Print s1 < s2                  ' Wahr
Debug.Print s2 < s3                  ' Wahr

Debug.Print "String mit Zahl vergleichen:"
Debug.Print i1 = s1                  ' Wahr
Debug.Print i1 < s3                  ' Fehler: Typen unverträglich!
End Sub
```

Wenn ein Double mit einem Single verglichen wird, runden VBA den Double für den Vergleich in einen Single. Wenn ein Wert vom Typ **Currency** mit einem **Single** oder **Double** verglichen wird, wandelt VBA den **Single** bzw. **Double** für den Vergleich in den Typ **Currency** um.

2.4.4 Boolesche Operatoren

Boolesche Ausdrücke sind Ausdrücke, die einen der beiden Werte **Wahr/True** oder **Falsch/False** ergeben. Beispiele dafür sind die Vergleiche im Code-Beispiel von Abschn. 2.4.3. Als Boolesche Operatoren bezeichnet man Operatoren, die auf Boolesche Ausdrücke angewendet werden. Man kann mit ihnen mehrere Boolesche Ausdrücke verknüpfen. Das Ergebnis einer Booleschen Operation ist ein Boolescher Wert.

Boolesche Operatoren gibt es in fast allen Programmiersprachen. Tab. 2.5 zeigt, wie man Boolesche Operatoren in VBA schreibt.

Der folgende Demo-Schnipsel zeigt eine mögliche Verwendung des Booleschen Operators **Or**.

```
Const MAX_TEILNEHMER As Integer = 5
Sub BooleDemo()
    Dim anmeldungen As Integer
    anmeldungen = 5 ' oder -1 oder 200 ...

    If anmeldungen > MAX_TEILNEHMER _
        Or anmeldungen < 0 Then
        MsgBox "Wert unzulässig!"
    End If
End Sub
```

Beim Formulieren von Bedingungen ist es wichtig, die Präzedenz der Booleschen Operatoren zu beachten: **And** bindet stärker als **Or**, analog zu „Punktrechnung vor Strichrechnung“. **Not** bindet am stärksten, analog zum Minuszeichen vor einer negativen Zahl. Komplexe Boolesche Formeln gliedert man am besten durch Klammern und macht sie dadurch auch leichter verständlich. Boolesche Ausdrücke dienen in Programmen vor allem dazu, den Ablauf des Programms zu steuern. Damit befassten sich die nächsten Abschnitte.

Tab. 2.5 Boolesche Operatoren in VBA

Zeichen	Funktion	Anwendbar für
Not	Nicht	Boolesche Ausdrücke
And	Und	Boolesche Ausdrücke
Or	Und/oder	Boolesche Ausdrücke
Xor	Entweder/oder	Boolesche Ausdrücke

2.5 Schleifen

Wenn keine besonderen Maßnahmen getroffen werden, führt der Interpreter die Befehle eines Programms in der Reihenfolge aus, in der sie im Programmcode stehen, also sequenziell. So lassen sich aber nur sehr einfache, unflexible Programme realisieren. Programmiersprachen bieten deshalb Sprachmittel, um den Programmablauf situationsabhängig, dass heißt, durch Daten zu steuern. Diese Sprachmittel bezeichnet man als Kontrollstrukturen.

Man spricht in diesem Zusammenhang vom „Kontrollfluss“. Der Kontrollfluss drückt aus, in welcher Reihenfolge die Befehle eines Programms bei einer Ausführung aufeinander folgen. Diese Reihenfolge kann von der Anordnung der Befehle im Quellcode abweichen und für verschiedene Programmausführungen, je nach Datenlage, unterschiedlich sein.

Grundsätzlich sind nur zwei Arten von Kontrollstrukturen nötig, um den Kontrollfluss in einem Programm zu steuern, nämlich Schleifen und Verzweigungen. Dieser Abschnitt befasst sich mit Schleifen in VBA. Manche Programmiersprachen, darunter VBA, bieten außerdem noch Sprungbefehle. In VBA verwendet man sie, um Laufzeitfehler behandelnd, siehe dazu Abschn. 4.1.

In der Programmierung ist eine Schleife eine Sequenz von Befehlen, die mehrfach durchlaufen wird. Wenn der letzte Befehl der Schleife abgearbeitet ist, kehrt der Kontrollfluss wieder an den Anfang der Schleife zurück und führt die Befehle (meist mit anderen Daten) erneut aus.

Formal besteht eine Schleife aus einem Schleifenrumpf und einer Schleifenbedingung. Als Schleifenrumpf bezeichnet man die Befehlsequenz, die mehrfach durchlaufen wird. Die Schleifenbedingung bestimmt, ob der Schleifenrumpf (nochmals) abgearbeitet oder die Schleife verlassen wird. Die Schleifenbedingung kann als Laufbedingung oder als Abbruchbedingung formuliert werden. VBA bietet mehrere Möglichkeiten, eine Schleife zu definieren.

2.5.1 ForNext-Schleifen

Eine ForNext-Schleife besitzt eine Zähl- oder Schleifenvariable, die bei jedem Schleifendurchlauf hochgezählt wird, und einen Endwert für die Schleifenvariable. Aus diesem Endwert ergibt sich eine Abbruchbedingung für die Schleife, denn vor jedem Schleifendurchlauf wird geprüft, ob die Schleifenvariable den Endwert überschreitet. Der Endwert und auch ein Startwert können als feste Werte oder als Variablen angegeben werden. Das folgende Code-Beispiel zeigt, wie man in VBA eine einfache ForNext-Schleife programmiert.

```
Sub ForNextDemo()
    Dim vektor(1 To 5) As Integer
    Dim i As Integer
```

```

For i = 1 To 5
    vektor(i) = i * 10
    Debug.Print i & ": " & vektor(i)
Next i
Debug.Print i & " Fertig!!"
End Sub

```

Die Schleifenvariable heißt hier `i`. Das Schlüsselwort `For` markiert den Start der Schleife. Der Ausdruck `i = 1 To 5` legt `i` als Schleifenvariable fest und definiert ihren Start- und Endwert. Der Befehl `Next i` erhöht die Schleifenvariable um 1 und markiert das Ende der Schleife. Die beiden Befehle zwischen `For` und `Next` bilden den Schleifenrumpf. Die Schleifenbedingung lautet hier implizit: „bis `i > 5`“.

2.5.2 DoLoop-Schleifen

Eine weitere Art von Schleifen in VBA sind DoLoop-Schleifen. Bei DoLoop-Schleifen formuliert man die Schleifenbedingung explizit mit einem Booleschen Ausdruck, wahlweise als Laufbedingung oder als Abbruchbedingung. Dadurch ergeben sich zwei Varianten von DoLoop-Schleifen. Das folgende Code-Beispiel zeigt, wie man in VBA eine einfache DoLoop-Schleife mit Laufbedingung programmiert.

```

Sub DoWhileLoopDemo()
Dim vektor(1 To 5) As Integer
Dim i As Integer

i = 1
Do While i <= 5
    vektor(i) = i * 10
    Debug.Print i & ": " & vektor(i)
    i = i + 1
Loop
Debug.Print i & " Fertig!!"
End Sub

```

Diese Schleife imitiert das Verhalten der ForNext-Schleife aus dem letzten Code-Beispiel. Das Schlüsselwort `Do` markiert den Start der Schleife. Der Ausdruck `While i <= 5` definiert eine Laufbedingung: Solange die Laufbedingung `i <= 5` zutrifft, wird der Schleifenrumpf ausgeführt. Das Schlüsselwort `Loop` markiert das Ende der Schleife. Die beiden Befehle zwischen `Do` und `Loop` bilden den Schleifenrumpf. Es ist wichtig, dass der Schleifenrumpf einen Befehl enthält, der eine Änderung der Schleifenbedingung bewirken oder erkennen kann. Sonst läuft die Schleife endlos oder doch sehr lange. Im Code-Beispiel wird dazu die Variable `i` hochgezählt.

Das nächste Code-Beispiel zeigt, wie man in VBA eine einfache DoLoop-Schleife mit Abbruchbedingung programmiert. Die Abbruchbedingung dieser Schleife prüft eine Eingabe eines Benutzers.

```
Sub DoUntilLoopDemo()
Dim abbruch As Integer

abbruch = vbNo
Do Until abbruch = vbYes
    abbruch = MsgBox("Aufhören?", vbYesNo)
    Debug.Print abbruch
Loop
Debug.Print "Fertig!!"
End Sub
```

MsgBox ist eine eingebaute VBA-Funktion. Sie erzeugt ein Dialogfenster wie in Abb. 2.5. Die MessageBox liefert eine Antwort eines Benutzers codiert als Zahl. Damit man sich diese Zahlen nicht merken muss und Programme leichter zu verstehen sind, hat VBA für die Zahlencodes Konstanten definiert. Solche VBA-Konstanten gibt es auch in weiteren Bereichen. Sie beginnen immer mit „vb“. Wenn man das Code-Beispiel

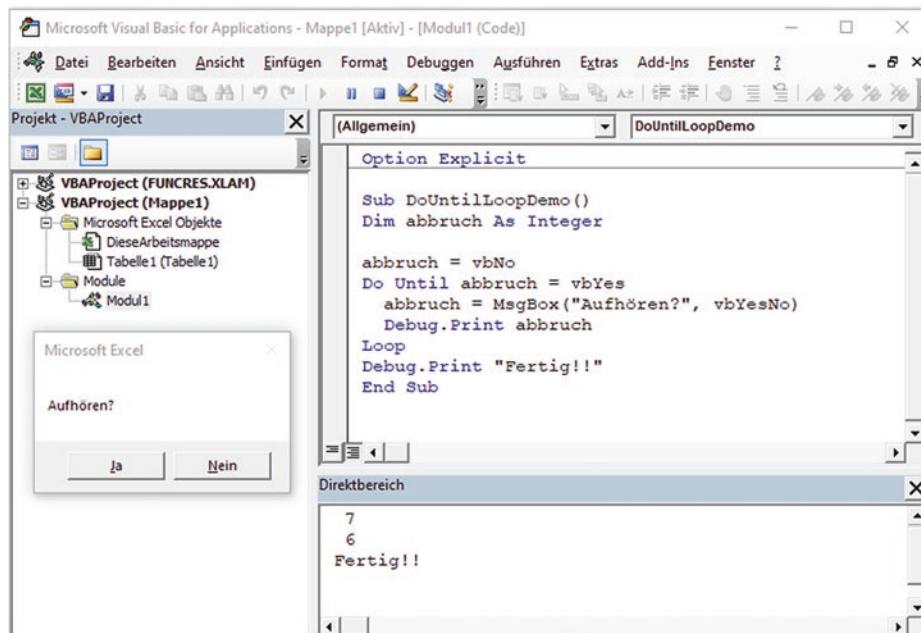


Abb. 2.5 Mit MsgBox ein Dialogfenster anzeigen

ausführt, sieht man, dass `vbYes` für die Zahl 6 steht und `vbNo` für die Zahl 7. Ein weiterer Zahlencode ist `vbYesNo`. Eine `MessageBox` mit Code `vbYesNo` hat einen Ja-Button und einen Nein-Button. Mit dem Code `vbOkOnly` würde man dagegen eine `MessageBox` mit einem einzigen Button „Ok“ erhalten.

Das nächste Code-Beispiel zeigt eine weitere Variante einer `DoLoop`-Schleife. Im Unterschied zur eben gezeigten Schleife prüft diese Schleife die Abbruchbedingung nicht vor dem ersten Schleifendurchlauf, sondern danach.

```
Sub DoLoopUntilDemo()
Dim abbruch As Integer

' abbruch = vbNo      hier nicht nötig
Do
    abbruch = MsgBox("Aufhören?", vbYesNo)
    Debug.Print abbruch
Loop Until abbruch = vbYes
Debug.Print "Fertig!!"
End Sub
```

Eine Laufbedingung mit `While` lässt sich ebenfalls vor oder hinter dem Schleifenrumpf abprüfen. Prüft man die Bedingung am Ende der Schleife, wird der Schleifenrumpf mindestens einmal abgearbeitet. Prüft man sie am Anfang der Schleife, wird der Schleifenrumpf möglicherweise gar nicht durchlaufen. Man wählt die Variante, die jeweils besser passt. Bei sehr einfachen Beispielen wie hier sind die Unterschiede unwesentlich.

2.5.3 ForEach-Schleifen

Neben `ForNext` ist `ForEach` eine weitere Möglichkeit, um Arrays zu durchlaufen. Statt mit dem Index durch die Positionen des Arrays zu gehen, verwendet man hier eine Element-Variable, die nach und nach jedes Element des Arrays aufnimmt. Der folgende Demo-Code zeigt dies. Für die Element-Variable verlangt VBA hier zwingend den Datentyp `Variant`.

```
Sub ForEachDemo()
Dim gruppe(1 To 26) As String
Dim kind As Variant

gruppe(1) = "Anna"
gruppe(2) = "Benno"
gruppe(3) = "Chris"
```

```
gruppe(4) = "Dani"  
gruppe(5) = "Emrah"  
gruppe(6) = "Finja"  
gruppe(26) = "Zora"  
  
For Each kind In gruppe  
    Debug.Print kind  
Next kind  
End Sub
```

Die Angabe der Element-Variablen nach `Next` (hier: `kind`) ist zwar optional, hilft aber bei langen oder geschachtelten Schleifen, den Überblick zu bewahren. `ForEach` funktioniert nicht nur für Arrays, sondern auch für weitere aufzählbare Datentypen und wird in den Beispielen dieses Buchs noch mehrmals benutzt.

2.5.4 Geschachtelte Schleifen

Wenn der Rumpf einer Schleife eine weitere Schleife enthält, spricht man von „geschachtelten Schleifen“. Der folgende Demo-Code füllt ein zweidimensionales Array mit Werten von 0 bis 11.

```
Sub ForNextInForNextDemo()  
Const anzZeilen As Integer = 3  
Const anzSpalten As Integer = 4  
Dim matrix(1 To anzZeilen, 1 To anzSpalten) As String  
  
Dim zeile As Integer  
Dim spalte As Integer  
Dim wert As Integer  
  
For zeile = 1 To anzZeilen  
    For spalte = 1 To anzSpalten  
        matrix(zeile, spalte) = wert  
        wert = wert + 1  
    Next spalte  
Next zeile  
  
Debug.Print matrix(1, 1)  
Debug.Print matrix(2, 2)  
Debug.Print matrix(3, 3)  
Debug.Print matrix(3, 4)  
End Sub
```

Wenn man lange und viele Schleifen ineinanderschachtelt, wird der Code schnell unübersichtlich. Als Faustregel gilt: zwei oder drei Verschachtelungsebenen sind noch beherrschbar, bei noch mehr Ebenen sollte man den Code mit Prozeduren oder Funktionen strukturieren. Abschn. 2.7 stellt diese vor.

2.6 Verzweigungen

„Wo soll's denn weiter gehen? — Kommt darauf an...“ Dafür gibt es in der Programmierung Verzweigungen. Sie leiten den Kontrollfluss abhängig von Bedingungen in verschiedene Richtungen.

2.6.1 IfThenElse-Verzweigung

Das folgende Code-Beispiel zeigt eine IfThenElse-Verzweigung:

```
Sub IfThenElseDemo()
    Dim a As Integer
    Dim b As Integer
    Dim abstand As Integer

    a = 60
    b = 90

    If a < b Then
        abstand = b - a
    Else
        abstand = a - b
    End If
    Debug.Print abstand
End Sub
```

2.6.2 IfThen-Verzweigung

Wenn im Else-Fall keine Befehle auszuführen sind, kann man ihn weglassen wie im folgenden Code-Beispiel:

```
Sub IfThenDemo()
    Dim a As Integer
    Dim b As Integer
    Dim abstand As Integer
```

```
a = 60
b = 90
abstand = a - b

If abstand < 0 Then
    abstand = abstand * (-1)
End If
Debug.Print abstand
End Sub
```

2.6.3 IfThenElseIfElse-Verzweigung

Verzweigungen mit mehr als zwei Alternativen codiert man mit ElseIf:

```
Sub IfThenElseIfElseDemo()
Dim land As String
land = "Schweiz"      ' Schweden Italien

If land = "Schweiz" Then
    Debug.Print "CHF"
ElseIf land = "Schweden" Then
    Debug.Print "SEK"
ElseIf land = "Großbritannien" Then
    Debug.Print "GBP"
Else
    Debug.Print "EUR"
End If
End Sub
```

2.6.4 SelectCase-Verzweigung

Wenn es sehr viele Alternativen gibt, ist das Konstrukt Select Case übersichtlicher. Select Case wertet einen Testausdruck aus, im folgenden Beispiel die Variable temperatur.

```
Sub SelectCaseDemo()
Dim temperatur As Single
temperatur = 3.1

Select Case temperatur
Case Is < 4
    Debug.Print "eisig"
```

```

    Debug.Print "Handschuhe erforderlich"
Case 4 To 8
    Debug.Print "kalt"
Case 8 To 15
    Debug.Print "kühl"
Case Else
    Debug.Print "warm"
End Select
End Sub

```

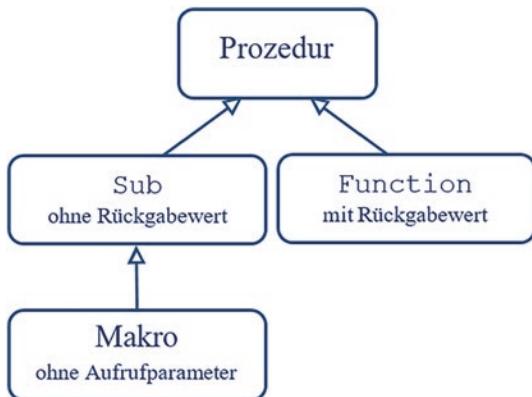
Die Case-Bedingungen werden von oben nach unten ausgewertet. Die Kontrolle fließt in den ersten zutreffenden Fall. Dann werden alle Befehle bis zur jeweils nächsten Case-Bedingung, oder bei der letzten Case-Bedingung bis zu End Select, ausgeführt. Wenn temperatur genau den Wert 8 hat, gibt das Programm also den Wert „kalt“ aus. Eine Case Else-Bedingung muss sinnvollerweise an letzter Stelle stehen. Sie ist optional. Wenn sie fehlt, endet Case Select gegebenenfalls ohne Aktivität, analog zu einer IfThen-Verzweigung ohne Else-Teil.

2.7 Prozeduren und Funktionen

Eine Prozedur ist eine Folge von Befehlen, die einen Namen hat und als Einheit ausgeführt wird. Eine Prozedur, die einen Wert berechnet und als Ergebnis zurückgibt, wird als Funktion bezeichnet. Prozeduren und Funktionen dienen dazu, Programme übersichtlich zu strukturieren. Die Bezeichnungen für Prozeduren und Funktionen und die Abgrenzung zwischen ihnen können sich von Programmiersprache zu Programmiersprache unterscheiden. In Visual Basic ist „Prozedur“ der Überbegriff für Sub und Function, wie in Abb. 2.6 dargestellt. Dabei ist eine Sub eine Prozedur ohne Rückgabewert und eine Function eine Prozedur mit Rückgabewert. Eine Sub ohne Aufrufparameter wird auch als Makro bezeichnet. Ein Makro kann direkt ausgeführt werden. Funktionen und Subs mit Aufrufparametern werden in anderen Subs aufgerufen, die ihre Aufrufparameter mit konkreten Werten belegen, mehr dazu erklärt Abschn. 2.7.1. In den folgenden Kapiteln wird der Begriff „Sub“ nicht nur als VBA-Schlüsselwort, sondern auch als fachlicher Begriff analog zu „Funktion“ verwendet.

Jede Prozedur oder Funktion soll eine überschaubare, in sich abgeschlossene Teilaufgabe innerhalb des gesamten Programms ausführen. Ein Programm mit Prozeduren und Funktionen zu strukturieren, bringt große Vorteile:

- Sie sind ein Mittel, um dieselbe Befehlssequenz an mehreren Stellen in einem Programm zu verwenden, ohne sie per Copy&Paste zu wiederholen. Dadurch sind Programme leichter zu pflegen, denn Änderungen und Fehlerkorrekturen bleiben auf eine Stelle im Programmcode beschränkt.

Abb. 2.6 Prozeduren in VBA

- Programme bleiben so kürzer und übersichtlicher.
- Programme sind leichter verständlich, weil die Bezeichnungen der Prozeduren und Funktionen Aufschluss darüber geben, wozu der in ihnen enthaltene Code dient. Zum Beispiel sollte eine Funktion, die das Maximum von zwei Werten ermittelt, idealerweise `maximum` oder `max` heißen.
- Prozeduren und Funktionen zerlegen eine komplexe Aufgabe in mehrere überschaubare Teilaufgaben, die sich leichter beherrschen lassen.
- Sie ermöglichen es, Programmierarbeit auf mehrere Personen zu verteilen.
- Sie lassen sich in weiteren Programmen wiederverwenden.

Prozeduren muss man zunächst definieren und kann sie dann in anderen Prozeduren aufrufen und verwenden. Die Definitionen von Funktionen und Subs stehen im Programmcode in beliebiger Reihenfolge untereinander. Die Definitionen können nicht ineinander geschachtelt werden.

2.7.1 Subs definieren und aufrufen

Das folgende Code-Beispiel zeigt die Definition einer Sub in VBA. Sie beginnt mit dem Schlüsselwort `Sub` und endet mit den Schlüsselwörtern `End Sub`. Auf das Schlüsselwort `Sub` folgen der Prozedurname und dann die Aufrufparameter. Aufrufparameter haben einen Datentyp und stehen in runden Klammern. Die darauf folgenden Befehle bilden den Prozedurrumpf. Sie werden beim Aufruf der Prozedur abgearbeitet. Der Prozedurname ist frei wählbar und sollte den Zweck der Prozedur deutlich machen. Für Prozedurnamen gelten die gleichen Regeln wie für Variablenbezeichner, siehe Abschn. 2.3.1.1.

Die Prozedur des folgenden Code-Beispiels hat den Namen `Bestaetigen` und zwei Aufrufparameter: `name` mit Datentyp `String` und `pers` mit Datentyp `Integer`. Sie setzt aus fixen Textteilen und wechselnden Parametern einen Info-Text zusammen und gibt ihn mit `Debug.Print` aus.

```
Sub Bestaetigen(name As String, pers As Integer)
Dim text As String

text = "Anmeldung von " & name & " mit " & pers & _
      " Personen bestätigt."
Debug.Print text
End Sub
```

Wie ihr Name sagt, erlauben es die Aufrufparameter einer Prozedur, sie für verschiedene Situationen zu parametrisieren, sodass die Prozedur flexibel verwendbar ist. Um eine Prozedur auszuführen, muss man sie aufrufen und ihre Aufrufparameter mit konkreten Werten füllen. Aufrufparameter ähneln Variablen. Beim Aufruf der Prozedur werden sie mit Werten belegt. Die Befehle im Prozedurrumpf greifen wie auf Variablen auf sie zu. Eine Prozedur kann im Prozedurrumpf auch eigene Variablen deklarieren und verwenden, so wie die Variable `text` in der Sub `Bestaetigen` im oben gezeigten Code-Beispiel.

Die konkreten Werte, mit denen die Aufrufparameter bei einem Aufruf der Prozedur belegt werden, bezeichnet man als Argumente des Prozedurauftrufs. In der Praxis verwendet man die beiden Begriffe Aufrufparameter und Argument aber häufig etwas unscharf und wie Synonyme.

Wenn eine Prozedur keine Aufrufparameter hat, bleiben die Klammern hinter dem Prozedurnamen in der Definition leer. Eine solche Sub bezeichnet man in VBA-Sprechweise auch als Makro. Eine Prozedur, die Aufrufparameter erfordert, lässt sich nicht direkt ausführen, sondern nur aus einer anderen Prozedur, die ihre Aufrufparameter mit konkreten Werten füllt. In einem größeren VBA-Projekt rufen sich Prozeduren mit und ohne Aufrufparameter gegenseitig auf. Die aufrufbare Top-Prozedur ist aber immer ein Makro.

Im folgenden Code-Beispiel ruft die Sub `ProzedurDemo` die Sub `Bestaetigen` vier Mal mit unterschiedlichen Argumenten auf. Beim ersten Aufruf sind der erste Aufrufparameter `name` mit dem Wert „Hans“ und der zweite Aufrufparameter `pers` mit dem Wert 4 belegt. Der zweite Aufruf der Sub erfolgt mit den Argumenten „Franz“ und 2 und so weiter.

```
Sub ProzedurDemo()
    Bestaetigen "Hans", 4
    Bestaetigen "Franz", 2
```

```
Bestaetigen "Silke", 3
Bestaetigen "Carol", 5
End Sub
```

Wenn der Kontrollfluss der aufrufenden Prozedur bei dem Aufruf angekommen ist, springt er in den Code der aufgerufenen Prozedur und arbeitet diesen bis zum Ende der Prozedur ab. Von dort springt er zurück in die aufrufende Prozedur und setzt deren Abarbeitung hinter dem Prozederaufruf fort.

2.7.2 Nice to know: Call by Reference versus Call by Value

In den Aufrufparametern übergibt die aufrufende Prozedur der aufgerufenen Prozedur Werte. Genauer gesagt übergibt sie diese in Form von Verweisen auf Variablen, also auf Speicherplätze. Die aufgerufene Prozedur kann die Werte an diesen Speicherplätzen lesen und sie kann auch neue Werte in diese Speicherplätze schreiben. Wenn die aufrufende Prozedur dann später auf Variablen zugreift, findet sie dort die von der aufgerufenen Prozedur modifizierten Werte. Diese Art der Parameterübergabe heißt „Call by Reference“. „Reference“ steht für „Verweis auf einen Speicherplatz“.

Bei der Definition einer Prozedur kann man für jeden Aufrufparameter einzeln festlegen, dass die Parameterübergabe anders erfolgen soll, nämlich so, dass die aufgerufene Prozedur keinen Verweis auf den Speicherplatz des Werts erhält, sondern den Wert direkt. VBA erstellt dazu intern eine Kopie des übergebenen Werts, mit dem die aufgerufene Prozedur arbeiten kann. Änderungen an der Kopie haben aber für die aufrufende Prozedur keine Wirkung. Aufrufparameter, die „By Value“ übergeben werden, kennzeichnet man in der Definition der Prozedur mit dem Schlüsselwort `ByVal`.

Parameter ohne diese Kennzeichnung werden standardmäßig „By Reference“ übergeben. Das bedeutet nicht, dass die aufgerufene Prozedur den Wert des Parameters tatsächlich ändert. Oft, wie in der Prozedur `Bestaetigen` des Code-Beispiels in Abschn. 2.7.2, tut sie dies nicht. Wenn eine Prozedur einen Wert für den aufrufenden Code ändern muss, kann man in der Definition dieser Prozedur das Schlüsselwort „`ByRef`“ explizit angeben und damit das Verhalten und die beabsichtigte Verwendung der Prozedur verdeutlichen.

2.7.3 Funktionen definieren und verwenden

Der wesentliche Unterschied zwischen einer `Sub` und einer `Function` besteht darin, dass eine Funktion einen Rückgabewert als Ergebnis hat. Während eine `Sub` dazu dient, Aktionen auszuführen, berechnet oder erstellt eine Funktion ein Ergebnis und liefert dieses an den aufrufenden Code zurück.

Die Definition einer Funktion beginnt mit dem Schlüsselwort `Function` und endet mit den Schlüsselwörtern `End Function`. Auf das Schlüsselwort `Function` folgen wie bei einer Sub der Name der Funktion und ihre Aufrufparameter in runden Klammern. Anders als bei einer Sub wird jetzt noch der Datentyp des Ergebnisses deklariert, das die Funktion zurückliefert.

Die darauf folgenden Befehle bilden den Funktionsrumpf. Hier wird auch die Rückgabe des Funktionsergebnisses programmiert. Ein Befehl zur Rückgabe eines Ergebnisses hat das Format `Funktionsname=Ergebnis`. Rückgabe-Befehle können beliebig oft und an beliebigen Stellen im Funktionsrumpf vorkommen. Wenn der Funktionsrumpf keinen Rückgabewert festlegt, liefert die Funktion ein Standard-Ergebnis, etwa 0 oder einen Leerstring, je nach Datentyp des Rückgabewerts.

Der folgende Demo-Code zeigt Definitionen von Funktionen, die sich auf die Illustration in Abb. 2.7 beziehen. Gleichzeitig zeigt der Demo-Code auch, wie man Funktionen benutzt. Das Makro `FunktionenDemo` verwendet die Funktion `Zylinderoberflaeche`, welche wiederum die übrigen drei Funktionen verwendet. Dabei ist auch zu sehen, wie der Rückgabewert einer Funktion einer Variablen zugewiesen oder der Funktionsaufruf direkt als Argument einer anderen Funktion eingesetzt werden kann.

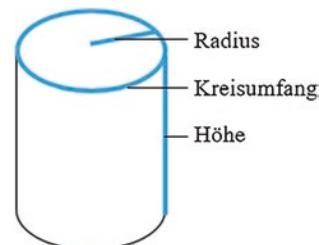
```
Function Kreisflaeche(radius As Double) As Double
Kreisflaeche = radius ^ 2 * 3.1416
End Function

Function Kreisumfang(radius As Double) As Double
Kreisumfang = 2 * radius * 3.1416
End Function

Function Rechteckflaeche(a As Double, b As Double) As Double
Rechteckflaeche = a * b
End Function

Function Zylinderoberflaeche(hoehe As Double, radius As Double) _
As Double
Dim deckel As Double
```

Abb. 2.7 Illustration „Zylinder“ für das Code-Beispiel zu Funktionen



```
Dim mantelflaeche As Double
deckel = Kreisflaeche(radius)
mantelflaeche = Rechteckflaeche(hoehe, Kreisumfang(radius))
Zylinderoberflaeche = 2 * deckel + mantelflaeche
End Function

Sub FunktionenDemo()
Debug.Print Zylinderoberflaeche(3, 4)
Debug.Print Zylinderoberflaeche(3, 5.5)
End Sub
```

2.7.4 Optionale und benannte Argumente

Wie von mathematischen Funktionen gewohnt, werden beim Aufruf einer Prozedur die Argumente den Aufrufparametern über die Position zugeordnet: Das erste Argument gehört zum ersten Aufrufparameter, das zweite Argument zum zweiten Aufrufparameter und so fort.

Die Zuordnung von Argumenten zu Aufrufparametern wird etwas komplizierter für Prozeduren mit optionalen Parametern, die es in VBA ebenfalls gibt. Beim Aufruf einer solchen Prozedur kann man ein Argument für einen optionalen Parameter angeben oder ihn auch weglassen. Dann arbeitet die Prozedur stattdessen mit einem fest vorgegebenen Standardwert, dem sogenannten Default.

Meist sind die obligatorischen Parameter vor den optionalen Parametern angeordnet. Wenn die Default-Werte passen, gibt man beim Prozederaufruf Argumente für die obligatorischen Parameter an, die wie gewohnt über die Position zugeordnet werden, und lässt die optionalen Parameter einfach weg. Wenn man jedoch für einzelne optionale Parameter statt der Default-Einstellungen eigene Werte festlegen möchte, muss der Prozederaufruf deutlich machen, welches Argument für welchen Parameter bestimmt ist. Hierfür bietet VBA zwei Schreibweisen. Der folgende Demo-Code zeigt sie am Beispiel der Funktion Mid. Diese Funktion schneidet einen Teilstring aus einem String. Sie hat drei Parameter:

- String ist der originale String, aus dem ein Teilstring ausgeschnitten wird,
- Start ist die Startposition, bei der der auszuschneidende Teilstring beginnt,
- Length ist die Anzahl der Zeichen des auszuschneidenden Teilstrings.

Der letzte Parameter Length ist optional. Er ist so voreingestellt, dass der ausgeschnittene Teilstring von der angegebenen Startposition bis zum Ende des originalen Strings reicht, wenn man keinen Wert für Length angibt. Der Demo-Code zeigt drei Möglichkeiten, Mid aufzurufen. Die jeweilige Rückgabe ist im Kommentar angegeben.

```
Sub OptArgDemo()
Dim orig As String: orig = "foobarbaz"

Debug.Print Mid(orig, 4, 6)                      ' barbaz
Debug.Print Mid(orig, 4)                          ' barbaz
Debug.Print Mid(String:=orig, Start:=4, Length:=3) ' bar
End Sub
```

Der dritte Aufruf von `Mid` arbeitet mit sogenannten benannten Argumenten. Hierbei gibt man den Namen eines Parameters an und ordnet ihm das für ihn bestimmte Argument mit den Zeichen `:=` zu.

Der Vorteil benannter Argumente wird deutlich bei Funktionen oder Subs, die viele Argumente akzeptieren. Beispielsweise hat die Prozedur `MsgBox` fünf Parameter, nämlich `Prompt`, `Buttons`, `Title`, `HelpFile` und `Context`. Nur der `Prompt` muss angegeben werden, die übrigen Parameter sind optional. Damit sind beispielsweise die Aufrufe im folgenden Demo-Code zulässig, die die Voreinstellung für den zweiten, optionalen Parameter `Buttons` beibehalten, aber einen eigenen Wert für den dritten, ebenfalls optionalen Parameter `Title` festlegen.

```
Sub OptArgDemo2()
Dim orig As String
orig = "Hallo"

MsgBox Prompt:=orig, Title:="Demo 1"
MsgBox orig, Title:="Demo 2"
MsgBox orig, , "Demo 3"
End Sub
```

Der dritte Aufruf von `MsgBox` zeigt eine neue Variante des Parameterübergabe. Hier wird die Position eines optionalen Parameters einfach übersprungen, indem man seine Position durch ein Komma markiert ohne einen Wert anzugeben.

`MsgBox` ist eigentlich eine Funktion, die einen Rückgabewert liefern kann wie in den Code-Beispielen in Abschn. 2.5.2. Das Code-Beispiel hier verwendet `MsgBox` wie eine Sub und ignoriert seinen Rückgabewert. Daher sind hier die Parameter wie bei einer Sub ohne Klammern anzugeben.

2.8 Module und Sichtbarkeit von Variablen und Prozeduren

Wenn ein Softwareprojekt wächst, kann es schnell unübersichtlich werden. Fehler-suchen, Anpassungen und Erweiterungen sind dann schwierig. Programmcode gut zu strukturieren ist eine der großen Herausforderungen in der Softwareentwicklung. VBA verwendet Module, um zusammengehörenden Code zu Einheiten zusammenzufassen

und gegenüber anderem Code abzuschotten. Ähnliche Mechanismen gibt es auch in anderen Programmiersprachen.

2.8.1 Module

VBA kennt verschiedene Arten von Modulen, die unterschiedlichen Zwecken dienen:

- UserForm-Module sind für den Code vorgesehen, der das Verhalten von UserForms (Formularen) realisiert. Kap. 9 zeigt dafür ein Beispiel.
- Für bestimmte Objekte einer Wirtsapplikation erstellt VBA automatisch Objektmodule, im Falle von Excel beispielsweise ein Objektmodul für die Arbeitsmappe und je ein Objektmodul für jedes Arbeitsblatt. Objektmodule sind der richtige Platz für Code, der das übliche Verhalten dieser Objekte modifiziert. Auch dazu folgt in Abschn. 13.5.2 ein Beispiel.
- Klassenmodule werden verwendet, um eigene Objektklassen zu definieren. Diese Programmietechnik wird im Folgenden nicht benötigt.
- Der gesamte sonstige Code gehört in Standard-Module.

Man kann beliebig viele Standard-Module anlegen. Jedes Modul soll Code für einen abgegrenzten Aufgabenbereich enthalten. Ein Standard-Modul hat zwei Abschnitte:

1. Variablen-Deklarationsabschnitt
2. Prozedur-Definitionsabschnitt

Der Variablen-Deklarationsabschnitt enthält Konstanten- und Variablen Deklarationen. Der Prozedur-Definitionsabschnitt enthält Sub- und Function-Prozeduren. Beide Abschnitte können auch leer bleiben.

2.8.2 Sichtbarkeit von Variablen und Konstanten

Module organisieren Code nicht nur optisch als Texte, sondern auch softwaretechnisch. Sie steuern Sichtbarkeit, das heißt, sie bestimmen, welche Teile eines Programms auf eine Variable, Konstante oder Prozedur zugreifen und sie verwenden können. Variablen und Konstanten unterscheiden sich hierbei nicht. Was im Folgenden für Variablen erklärt wird, gilt auch für Konstanten.

In VBA können Variablen an zwei Stellen deklariert werden:

- im Variablen-Deklarationsabschnitt eines Moduls und
- innerhalb einer Prozedur, also im Rumpf einer Prozedurdefinition zwischen Sub und End Sub oder Function und End Function.

Innerhalb einer Prozedur deklarierte Variablen heißen „auf Procedurebene deklarierte Variablen“ und sind sichtbar innerhalb der Prozedur. Das heißt, dass nur Befehle im Rumpf dieser Prozedur ihren Wert lesen oder verändern können. Im Variablen-Deklarationsabschnitt eines Moduls deklarierte Variablen heißen „auf Modulebene deklariert“ und sind auf Modulebene sichtbar. Alle Prozeduren im selben Modul können ihren Wert lesen oder verändern.

Man kann denselben Variablenbezeichner innerhalb eines Moduls mehrfach verwenden. Es ist zum Beispiel sinnvoll, Laufvariablen von ForNext- oder ForEach-Schleifen immer gleich zu benennen, wenn diese Schleifen dieselben Strukturen durchlaufen. Wenn sich die Sichtbarkeitsbereiche von Variablen überlappen, überdeckt (englisch „to shadow“) die Variable im engeren Sichtbarkeitsbereich die Variable im äußeren Sichtbarkeitsbereich. Das folgende Code-Beispiel demonstriert dies. Hier ist eine Variable mit dem Bezeichner a1 auf Modulebene definiert und weitere Variable mit dem gleichen Bezeichner in der Prozedur ProzedurVarSetzen. Die Prozedur ProzedurVarSetzen kann als einzige auf „ihre“ Variable a1 zugreifen. Alle anderen Prozeduren des Beispiels sehen die Variable a1, die auf Modulebene deklariert ist. Wenn man die Prozedur SichtbarDemo ausführt, erzeugt sie die Ausgaben, die als Kommentare im Code-Beispiel stehen.

```
Private a1 As String

Sub ModulVarSetzen()
    a1 = "Modulebene"
    Debug.Print a1
End Sub

Sub ProzedurVarSetzen()
    Dim a1 As String
    a1 = "Procedurebene"
    Debug.Print a1
End Sub

Sub ModulVarLesen()
    Debug.Print a1
End Sub

Sub SichtbarDemo()
    ' Ausgabe:
    Debug.Print a1
    ModulVarSetzen
    ' beim 1. Ausführen "", danach "Modulebene"
    ProzedurVarSetzen
    ' "Modulebene"
    ModulVarLesen
    ' "Modulebene"
    Debug.Print a1
    ' "Modulebene"
End Sub
```

Dieses Beispiel ist nicht zur Nachahmung empfohlen: Besser, weil klarer, wären unterschiedliche Variablenbezeichner auf Modul- und auf Prozedurenbene. Vor allem aber ist `a1` als Variablenbezeichner sehr nichtssagend und für eine Variable auf Modulebene keinesfalls passend.

Um Variablen modulübergreifend sichtbar zu machen, deklariert man sie auf Modulebene und mit dem Schlüsselwort `Public` statt `Dim`. Die entsprechende Deklaration für Konstanten ist `Public Const`. Der folgende Code-Schnipsel zeigt Beispiele:

```
Public Const INFO_TEXT As String = "Im ganzen Projekt verwendbar"  
Public AnzahlAnmeldungen As Long
```

Public Variablen und Konstanten ermöglichen gemeinsame Datennutzung und Informationsaustausch zwischen Modulen, schwächen zugleich aber die Modularität des Projekts. Um stabilen und beherrschbaren Code zu erhalten, deklariert man Variablen immer mit dem engsten noch ausreichenden Sichtbarkeitsbereich.

Das Gegenstück zum Schlüsselwort `Public` lautet `Private`. Ohne ausdrückliche Angabe eines dieser Schlüsselwörter sind Variablen und Konstanten standardmäßig `Private`. Wenn man verdeutlichen möchte, dass Variablen oder Konstanten nur innerhalb des Moduls sinnvoll nutzbar sind, deklariert man sie explizit als `Private`.

Im Zusammenhang mit Modulen und Sichtbarkeit steht auch das Konzept der Gültigkeit von Variablen. Letztendlich geht es dabei darum, wie lange der Speicherplatz für eine Variable reserviert bleibt. Auf Prozedurebene deklarierte Variablen werden angelegt, wenn die Prozedur startet. Wenn sie abgearbeitet ist, werden ihre Variablen wieder vernichtet und der für sie reservierte Speicherplatz wird für andere Zwecke freigegeben. Auf Modulebene deklarierte Variablen bleiben dagegen so lange gültig wie das Modul. Wenn man also die Sub `SichtbarDemo` aus dem Code-Beispiel oben ein zweites Mal ausführt, liefert bereits der erste `Debug.Print`-Befehl die Ausgabe „Modulebene“.

2.8.3 Sichtbarkeit von Prozeduren

Prozeduren können auf Projektebene oder auf Modulebene definiert werden. Mit `Sub` oder `Function` definierte Prozeduren sind im ganzen VBA-Projekt sichtbar. Um ihre Sichtbarkeit auf das Modul einzuschränken, kann man das Schlüsselwort `Private` einsetzen. Dies ist zum Beispiel sinnvoll, wenn Prozeduren sehr spezielle Hilfsaufgaben ausführen, die in einem anderen Kontext nicht sinnvoll sind, oder wenn sie auf nur lokal verfügbare Variablen zugreifen. Der folgende Code-Schnipsel zeigt die Schreibweise.

```
Private gruppe(1 To 26) As String  
  
Private Sub Initialisieren()  
gruppe(1) = "Anna"
```

```
gruppe(2) = "Benno"  
gruppe(3) = "Conrad"  
End Sub
```

2.9 Fazit

Moderne Programmiersprachen bauen auf verschiedene Prinzipien, damit der entwickelte Programmcode verständlich, wartbar und in seiner Komplexität beherrschbar bleibt. Wenn eine Programmiersprache Kontrollstrukturen wie Schleifen und Verzweigungen bietet und die Definition von Prozeduren und zusammengesetzten Datentypen erlaubt, spricht man von Strukturierter Programmierung. Erste strukturierte Programmiersprachen sind in den 1960er Jahren entstanden. Etwas später, in den 1970er Jahren, kam das Prinzip der Modularisierung hinzu. Modulare Programmierung bedeutet, dass Code auf spezialisierte, gegeneinander abgegrenzte Module aufgeteilt werden kann, die sich wiederverwenden und austauschen lassen. Aus den vorangegangenen Abschnitten wird klar, dass es sich bei VBA um eine strukturierte und modulare Programmiersprache handelt.

Seit etwa den 1980er Jahren ist die objektorientierte Programmierung bekannt und wird sehr erfolgreich eingesetzt, um Softwaresysteme mit vielfältigen Interaktionsmöglichkeiten und komplexen Abläufen zu entwickeln. Das folgende Kapitel befasst sich mit den objektorientierten Aspekten von VBA.



Objektbasiertes Programmieren

3

Inhaltsverzeichnis

3.1	Klasse und Objekt	44
3.2	Der Objekttyp Collection	45
3.3	Befehle und Schreibweisen zum Umgang mit Objekten	46
3.3.1	Objektvariable und Set-Operator	46
3.3.2	Standard-Eigenschaft und Standard-Methode	46
3.3.3	With-Schreibweise	47
3.4	Objekte aus dem Objektmodell von Excel	47
3.4.1	Das Application-Objekt	48
3.4.2	Objekttypen für Arbeitsmappen, Arbeitsblätter und Zellen	49
3.4.3	Collections im Objektmodell	49
3.4.4	Befehle zum Umgang mit Excel-Objekten	50
3.4.5	Kurzformen und Voreinstellungen im Excel-Objektmodell	51
3.5	Demo-Anwendung „Verlinkte Datenblätter“	52
3.5.1	Sub Main und Konstantendeklaration	54
3.5.1.1	Deaktivieren der Bildschirmanzeige	54
3.5.1.2	Speichern der Arbeitsmappe	54
3.5.2	Sub BlaetterEinrichten	54
3.5.2.1	Einfügen eines Arbeitsblatts an letzter Stelle	56
3.5.2.2	Zellen formatieren	56
3.5.2.3	Farben definieren	56
3.5.2.4	Hyperlinks einfügen	57
3.5.3	Sub StartEinrichten	57
3.5.3.1	Arbeitsblatt formatieren	58
3.5.3.2	Einfügen einer Schaltfläche	59
3.5.3.3	Arbeitsblatt aktivieren	59
3.5.3.4	Gitternetzlinien ausblenden	59
3.5.4	Sub Aufraeumen	59
3.6	Fazit und Ausblick	60

Objektorientierte Programmierung ist ein Programmierparadigma, das heißtt, eine bestimmte Art, an die Aufgabenstellung der Programmierung heranzugehen und Software aufzubauen. Charakteristisch für objektorientierte Programmierung ist, dass sie Software als ein System von Klassen und Objekten auffasst, wobei sich die Objekte aus den Klassen ableiten. Objektorientiertes Programmieren bedeutet, Klassen zu entwerfen und dazu eine Logik, nach der bei der Ausführung des Programms aus den Klassen Objekte erzeugt werden, sodass die Objekte die Funktionalität der entwickelten Software realisieren.

Wenn man VBA zur Automatisierung von Anwendungssystemen einsetzt, sind viele gut designte Klassen bereits vorhanden. Sie stammen aus den Anwendungssystemen und stellen deren Komponenten und Funktionen dar. Wenn man einige grundlegende Prinzipien und Klassen kennt, lassen sich diese sehr einfach und intuitiv in Programmen einsetzen. Dieses Kapitel stellt die wichtigsten Befehle und Mechanismen zum Umgang mit Objekten in VBA vor und zeigt sie am Beispiel von Excel in Aktion.

3.1 Klasse und Objekt

Die zentralen Begriffe der objektorientierten Programmierung sind „Objekt“ und „Klasse“. Verschiedene objektorientierte Programmiersprachen fassen den Begriff „Klasse“ etwas unterschiedlich auf und bieten auch unterschiedliche Möglichkeiten, mit Klassen umzugehen und sie einzusetzen. Alle teilen jedoch das Grundverständnis, dass eine Klasse eine Schablone oder einen Bauplan für Objekte definiert, aus dem man neue Objekte erzeugen kann. Ein aus dem Bauplan einer Klasse abgeleitetes Objekt bezeichnet man als eine Instanz dieser Klasse. Ein Objekt besitzt die Felder und Methoden, die im Bauplan der Klasse vorgegeben sind.

- Felder, auch als Eigenschaften bezeichnet, sind Datenspeicher. Sie speichern Informationen über den Zustand des Objekts. Bei den Informationen kann es sich um einfache Datenelemente wie beispielsweise Strings oder numerische Werte handeln, oder auch um andere Objekte oder Listen von Objekten.
- Methoden sind Aktionen, die ein Objekt ausführen kann.

Zusammenfassend bezeichnet man die Felder und Methoden eines Objekts auch als seine Glieder, Member oder Elemente. Objekte, die nach demselben Bauplan erzeugt wurden, besitzen die gleichen Felder und Methoden, wobei die Felder der einzelnen Objekte mit jeweils unterschiedlichen Werten gefüllt sein können. Man bezeichnet den gleichartigen Aufbau dieser Objekte als Objekttyp, analog zum Datentyp bei einfachen Daten.

3.2 Der Objekttyp Collection

Die meisten Klassen, mit denen man in VBA-Programmen umgeht, stammen aus den Anwendungssystemen, die mit VBA angesteuert werden. Collection ist eine Klasse, die zu VBA selbst gehört. Eine Instanz dieser Klasse, also ein Collection-Objekt, dient zur Verwaltung einer Liste von Elementen. Die Collection ähnelt einem Array, ist aber mächtiger und auch komfortabler zu nutzen. Während die Größe eines Arrays deklariert werden muss, kann eine Collection beliebig wachsen und schrumpfen. Die Collection wird hier deshalb ausführlich vorgestellt, weil sie bei der Automatisierung von Anwendungssystemen sehr häufig vorkommt und oft gebraucht wird. Der folgende Demo-Code zeigt, wie man eine Collection einsetzen kann.

```
Sub CollectionDemo()
Dim chefs As Collection
Dim chef As Variant

Set chefs = New Collection
chefs.Add "Erwin", "Co-Founder1"
chefs.Add "Darwin", "Co-Founder2"
chefs.Add "Franzi", "CFO"
chefs.Add "Uli", "CEO"
chefs.Add "Kim", "CIO"

Debug.Print chefs.Count
Debug.Print chefs(1)
Debug.Print chefs(5)
Debug.Print chefs("CIO")                                ' Ausgabe:
                                                       ' 5
                                                       ' Erwin
                                                       ' Kim
                                                       ' Kim

chefs.Remove "Co-Founder1"
chefs.Add "Sam", "Big Boss", before:="CFO"
Debug.Print "----"                                     ' ---
For Each chef In chefs
    Debug.Print chef                                    ' Darwin Sam Franzi Uli Kim
Next chef
Set chefs = Nothing
End Sub
```

Der Befehl `Dim chefs As Collection` deklariert eine Variable für ein Collection-Objekt. Der Befehl `Set chefs = New Collection` erzeugt eine Instanz der Klasse Collection, also ein Collection-Objekt, und weist es der Variablen `chefs` zu. Der letzte Befehl der Sub, `Set chefs = Nothing`, zerstört dieses Objekt wieder und gibt den dafür reservierten Speicherplatz für andere Zwecke frei.

Ein Collection-Objekt besitzt ein Feld Count, in dem es die Anzahl seiner Listenelemente speichert. Das Objekt hält sein Feld Count automatisch aktuell, wenn Listenelemente dazukommen oder wegfallen. Die Methode Add fügt der Liste ein neues Element hinzu. Diese Methode hat vier Argumente. Das erste, obligatorische Argument ist das Element, das der Collection hinzugefügt wird. Die übrigen drei Argumente sind optional. Das zweite Argument von Add ist ein Schlüssel oder key, mit dem man das hinzugefügte Element identifizieren und wieder abrufen kann. Wenn man keinen Schlüssel angibt, lässt sich das Element nur über seinen Index identifizieren, also über seine Position in der Liste. Das dritte Argument before gibt an, wo das neue Element in die Liste eingefügt werden soll. Man kann hier einen Schlüssel oder einen Index angeben. Das vierte Argument after funktioniert analog zu before. Ohne before oder after stellt die Methode Add das Element an das Ende der Liste. Die Methode remove entfernt ein Element. Auch hier kann man das Element mit seinem Index identifizieren oder mit seinem Schlüssel, sofern es einen Schlüssel hat.

Eine Collection lässt sich mit einer ForEach-Schleife durchlaufen. Wenn die Collection Elemente mit einem einfachen Datentyp speichert, muss die Schleifenvariable (im Demo-Code chef) von Typ Variant sein. Wenn eine Collection Objekte als Elemente besitzt, kann man die Schleifenvariable mit dem passenden Objekttyp deklarieren. Die Demo-Anwendung dieses Kapitels zeigt dazu Beispiele. Im Folgenden wird der Begriff „Collection“ außer als VBA-Schlüsselwort Collection auch als Fachausdruck verwendet ähnlich zu „Array“ und wird dann in normaler Schrift gedruckt.

3.3 Befehle und Schreibweisen zum Umgang mit Objekten

Auf die Felder und die Methoden eines Objekts greift man in VBA mittels der Punkt-Schreibweise zu, die auch in anderen objektorientierten Programmiersprachen geläufig ist.

3.3.1 Objektvariable und Set-Operator

Objekte kann man in Variablen zugreifbar machen. Eine Variable, die ein Objekt oder genau genommen einen Verweis auf ein Objekt speichert, bezeichnet man als Objektvariable. Man deklariert sie mit dem Objekttyp des Objekts. Der Demo-Code zur Collection in Abschn. 3.2 deklariert eine Objektvariable chefs für den Objekttyp Collection. Für Objektvariablen hat VBA einen speziellen Zuweisungsoperator mit dem Schlüsselwort Set. Er wird auch im oben gezeigten Demo-Code eingesetzt.

3.3.2 Standard-Eigenschaft und Standard-Methode

Der Abruf von Elementen aus einer Collection verdient einen genaueren Blick. Die Befehle chefs("CIO") und chefs(5) sehen zwar syntaktisch aus wie

Funktionsaufrufe, doch `chefs` ist kein Funktionsbezeichner, sondern bezeichnet ein Objekt. Worum handelt es sich hier also?

In VBA können Objekte eine Standard-Eigenschaft oder -Methode besitzen, die bei der Entwicklung der Klasse des Objekts festgelegt wurde. Wenn VBA-Code ein Objekt anspricht, ohne eine Eigenschaft oder Methode anzugeben, greift der Interpreter auf die Standard-Eigenschaft oder Standard-Methode zu. Collection-Objekte besitzen eine Methode `Item`, die man mit einem Index oder einem Schlüsselwert als Argument aufrufen kann. Sie ist in der Collection als Standard-Methode voreingestellt. Die Ausdrücke `chefs("CIO")` und `chefs(5)` sind Kurzformen für `chefs.Item("CIO")` beziehungsweise `chefs.Item(5)`.

3.3.3 With-Schreibweise

Wenn mehrere Eigenschaften oder Methoden eines Objekts angesprochen werden müssen, gliedert die With-Schreibweise den Code und macht ihn kürzer und übersichtlicher. Dazu wird das Objekt mit dem Schlüsselwort `With` angegeben. In allen Befehlen bis zum `End With`-Ausdruck kann jetzt ein Punkt das Objekt vertreten. Der folgende Code-Schnipsel zeigt ein Beispiel.

```
Sub WithDemo ()  
Dim chefs As Collection  
Dim chef As Variant  
  
Set chefs = New Collection  
With chefs  
    .Add "Erwin", "Co-Founder1"  
    .Add "Darwin", "Co-Founder2"  
    .Add "Franzi", "CFO"  
    .Add "Uli", "CEO"  
    .Add "Kim", "CIO"  
    Debug.Print .Count      ' 5  
End With  
Set chefs = Nothing  
End Sub
```

3.4 Objekte aus dem Objektmodell von Excel

Die Komponenten einer Anwendungssoftware, die sich mit VBA ansteuern lässt, erscheinen in VBA als Objekte. Die Gesamtheit aller Objekttypen, die eine Software für die Manipulation mit VBA offenlegt, bilden das Objektmodell dieser Anwendung.

3.4.1 Das Application-Objekt

Ein besondere Rolle spielt dabei der Objekttyp `Application`. Er repräsentiert das Anwendungssystem als Ganzes. Aus Sicht von VBA ist ein laufendes Excel eine Instanz der `Application`-Klasse aus dem Objektmodell von Excel, also ein Excel-Objekt. Auch seine Komponenten wie Arbeitsmappen, Arbeitsblätter usw. sind Instanzen von weiteren Klassen, die im Objektmodell von Excel modelliert sind. Wird in Excel eine neue Arbeitsmappe erzeugt, erscheint sie in VBA als ein neues Arbeitsmappen-Objekt. Wird umgekehrt durch VBA-Code ein neues Arbeitsmappen-Objekt erzeugt, erscheint auch eine neue Arbeitsmappe in Excel.

Das `Application`-Objekt speichert Informationen über den aktuellen Zustand der Anwendung und Einstellungen, die die gesamte Anwendung betreffen. Solche Einstellungen sind zum Beispiel `ScreenUpdating` und `DisplayAlerts`, deren Funktion in den Code-Beispielen dieses Kapitels erklärt wird. In seinen Feldern `ActiveWorkbook`, `ActiveSheet` und `ActiveCell` speichert das `Application`-Objekt, welche der Komponenten von Excel gerade im Vordergrund stehen beziehungsweise angewählt sind.

Das `Application`-Objekt besitzt außerdem Felder für den Zugriff auf seine Komponenten. Diese wiederum haben Felder für den Zugriff auf ihre Komponenten und so weiter. Excel und andere Anwendungssysteme, die sich mit VBA automatisieren lassen, zeigen sich in VBA als eine Struktur von Objekten mit einem `Application`-Objekt als Top-Element und Einstiegspunkt. Abb. 3.1 verbildlicht einen unvollständigen

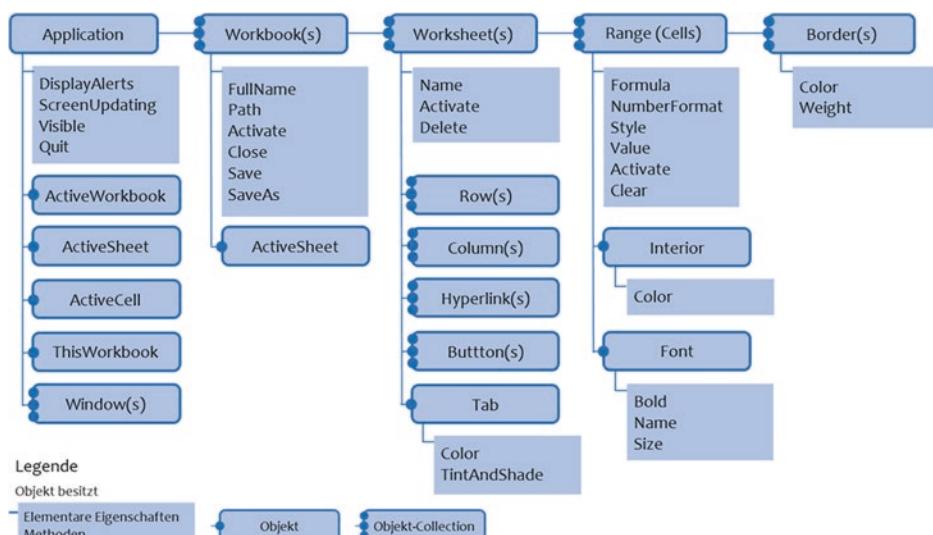


Abb. 3.1 Ausschnitt aus dem Objektmodell von Excel

Ausschnitt aus dem Objektmodell von Excel. Einzelne Objekte werden im Folgenden näher erläutert und in Code-Beispielen verwendet.

3.4.2 Objekttypen für Arbeitsmappen, Arbeitsblätter und Zellen

Die wichtigsten Objekttypen aus dem Objektmodell der Anwendung Excel neben Application sind Workbook, Worksheet und Range.

Arbeitsmappen sind in VBA als Objekte des Objekttyps Workbook repräsentiert. Dieser Objekttyp besitzt unter anderem Felder für den Namen der Arbeitsmappe und für ihren Dateipfad, Methoden zum Aktivieren, Schließen und Speichern der Arbeitsmappe und ein Feld Worksheets, welches ihre Arbeitsblätter verwaltet.

Ein Arbeitsblatt zeigt sich in VBA als ein Objekt des Typs Worksheet. Jedes Worksheet-Objekt besitzt unter anderem ein Feld Name, das den Namen des Arbeitsblatts speichert, und ein Feld Cells, das auf die Zellen des Arbeitsblatts verweist. Methoden eines Worksheet-Objekts sind Activate und Delete. Mit seiner Methode Activate bringt sich ein Arbeitsblatt in den Vordergrund, mit seiner Methode Delete zerstört es sich.

Ein weiterer zentraler Objekttyp ist Range. Ein Range-Objekt bildet Zellenbereiche ab, wobei ein Zellenbereich oft auch nur eine einzelne Zelle umfasst. Ein Range-Objekt hat unter anderem die Felder Value und Formula. Zu seinen Methoden gehört Clear und ebenfalls eine Methode Activate, mit der sich ein Zellenbereich-Objekt aktiviert, das heißt, anwählt.

3.4.3 Collections im Objektmodell

Das Objektmodell bildet die Beziehungen zwischen einer Arbeitsmappe und ihren Arbeitsblättern mit einer speziellen Konstruktion ab, die auch an anderen Stellen in Objektmodellen oft vorkommt. Deshalb wird sie jetzt genauer betrachtet.

Der Objekttyp Workbook besitzt ein Feld Worksheets, also einen Speicherplatz ähnlich einer Variablen. Dieses Feld Worksheets verweist auf ein Objekt des Typs Worksheets, das im Grunde eine spezielle Art Collection ist. Ihre Elemente sind Objekte des Typs Worksheet, also Arbeitsblätter. Ein Worksheets-Objekt besitzt Felder und Methoden, die für eine Collection typisch sind, darunter das Feld Count und die Methode Item zum Abruf von Elementen. Außerdem besitzt es spezialisierte Methoden zum Umgang mit Objekten des Typs Worksheet. Zu diesen gehören die Methoden Copy und Move zum Kopieren und zum Verschieben von Arbeitsblättern sowie die Methode Add, die ein Worksheet-Objekt nicht nur in die Collection einfügt, sondern es auch neu erzeugt. Die Worksheets-Collection einer Arbeitsmappe beinhaltet alle Arbeitsblätter dieser Arbeitsmappe. Sie können per Index oder Schlüssel

in der Worksheets-Collection identifiziert werden. Als Schlüssel dient dabei der Name eines Arbeitsblatts.

Nach diesem Muster werden noch viele weitere Objekttypen im Excel-Objektmodell durch eine Collection erzeugt und verwaltet. Die Zellen eines Arbeitsblatts sind als Range-Objekte in der Collection Cells eines Worksheet-Objekts hinterlegt. Arbeitsmappen sind als Workbook-Objekte in der Collection Workbooks des Application-Objekts hinterlegt. Abb. 3.1 repräsentiert diese und weitere Collection-Objekte mit den von ihnen verwalteten Objekttypen jeweils in einem gemeinsamen Symbol.

Es ist in diesem Zusammenhang auch hilfreich, sich klarzumachen, dass jedes Arbeitsblatt und jede Zelle nur einmal als Objekt (das heißt Datenstruktur) im Arbeitsspeicher des Computers existiert, aber von mehreren Stellen aus referenziert werden kann, zum Beispiel aus einer Worksheets- oder Cells-Collection, aus den Feldern ActiveSheet oder ActiveCell des Application-Objekts und eines Workbook-Objekts und auch von Objektvariablen.

3.4.4 Befehle zum Umgang mit Excel-Objekten

Der folgende Demo-Code zeigt einige typische Befehle für den Umgang mit Excel-Objekten in VBA. Diese erzeugen ein neues Arbeitsblatt, befüllen es mit Werten und zerstören es schließlich wieder.

```
Sub ArbeitsblattDemo()
    Dim NeuesBlatt As Worksheet

    Set NeuesBlatt = ThisWorkbook.Worksheets.Add
    NeuesBlatt.Name = "Demo"
    NeuesBlatt.Cells(1, 1).Value = "Hallo Welt"
    NeuesBlatt.Cells(2, 1).Value = 7
    NeuesBlatt.Cells(2, 2).Value = 6
    NeuesBlatt.Cells(2, 3).Formula = "=A2*B2"
    MsgBox NeuesBlatt.Cells(2, 3).Value
    MsgBox NeuesBlatt.Name & " wird wieder gelöscht...", vbOKOnly, "Demo"
    NeuesBlatt.Delete
End Sub
```

Der Demo-Code deklariert eine Objektvariable mit dem Bezeichner „NeuesBlatt“ für ein Arbeitsblatt. ThisWorkbook ist ein Objekt vom Typ Workbook, nämlich die Arbeitsmappe, in deren VBA-Projekt der Demo-Code liegt. Im Ausdruck ThisWorkbook.Worksheets.Add wird die Add-Methode des Worksheets-Objekts dieser Arbeitsmappe aufgerufen, um ein neues Arbeitsblatt zu erstellen. Es wird sofort der Objektvariablen NeuesBlatt zugewiesen und in den folgenden Befehlen über diese

Objektvariable angesprochen. Zuerst wird so die Eigenschaft Name des Worksheet-Objekts mit dem Wert „Demo“ belegt.

Die darauffolgenden vier Befehle des Demo-Skripts greifen auf die Felder Value oder Formula von Range-Objekten zu. Die Ausdrücke NeuesBlatt.Cells(2, 3).Value und NeuesBlatt.Name rufen einen Wert aus einem Feld eines Range-Objekts beziehungsweise eines Worksheet-Objekts ab. Der Befehl NeuesBlatt.Delete ruft eine Methode eines Worksheet-Objekts auf.

Während dieser Demo-Code ausgeführt wird, sind im Hintergrund einige Automatismen der beteiligten Objekte am Werk. Beispielsweise aktualisiert das Application-Objekt automatisch seine ActiveSheet-Eigenschaft, ohne dass dafür ein ausdrücklicher VBA-Befehl nötig wäre. Das Range-Objekt, das eine Formel enthält, aktualisiert sein Feld Value automatisch, und beim Zerstören des Arbeitsblatts wird automatisch ein Dialog mit einer Sicherheitsabfrage angezeigt. Dieses selbstgesteuerte Verhalten von Objekten macht einen wesentlichen Teil der Stärke des objektorientierten Programmierparadigmas aus.

3.4.5 Kurzformen und Voreinstellungen im Excel-Objektmodell

Auch im Excel-Objektmodell gibt es viele Standard-Eigenschaften und Voreinstellungen, darunter diese:

- Das Application-Objekt muss nicht angegeben werden.
- Wenn ein Arbeitsmappen-Objekt benötigt wird, aber nicht genannt ist, wirken die VBA-Befehle auf das ActiveWorkbook.
- Wenn kein Arbeitsblatt angegeben wird, greift der Code auf das ActiveSheet zu.
- Ein neu erstelltes Arbeitsblatt ist automatisch aktiviert.
- Die Standard-Eigenschaft des Range-Objekts ist Value.

Wenn man diese Voreinstellungen verwendet, kann man den oben gezeigten Demo-Code auch so schreiben:

```
Sub ArbeitsblattDemoKurz()
    Worksheets.Add
    ActiveSheet.Name = "Demo"
    Cells(1, 1) = "Hallo Welt"
    Cells(2, 1) = 7
    Cells(2, 2) = 6
    Cells(2, 3).Formula = "=A2*B2"
    MsgBox Cells(2, 3)
    MsgBox ActiveSheet.Name & " wird wieder gelöscht...", vbOKOnly, "Demo"
    ActiveSheet.Delete
End Sub
```

Der Code wirkt damit sehr übersichtlich. Durch den expliziten und impliziten Zugriff auf ActiveSheet ist er aber auch sehr situationsabhängig und darum ist die Kurzschreibweise nicht gut geeignet für VBA-Projekte, die mit mehreren Arbeitsblättern oder Arbeitsmappen umgehen.

3.5 Demo-Anwendung „Verlinkte Datenblätter“

Die Demo-Anwendung „Verlinkte Datenblätter“ erzeugt eine Menge von Arbeitsblättern, richtet diese ein und verlinkt sie. Sie demonstriert damit die bisher beschriebenen Programmiertechniken und zeigt weitere Möglichkeiten des Excel-Objektmodells. Das Anwendungsbeispiel arbeitet mit einer Liste mit Personendaten, doch auch Produktblätter, Datenerfassungsbögen, Fehlerberichte und vieles mehr lassen sich auf diese Weise übersichtlich darstellen.

Die Daten für die Demo-Anwendung stehen als Liste in den Zeilen eines Excel-Arbeitsblatts „sportgruppe“ wie in Abb. 3.2 links. Davon ausgehend erzeugt die Demo-Anwendung ein Start-Arbeitsblatt wie in Abb. 3.2 rechts und für jeden Datensatz ein neues Daten-Arbeitsblatt wie in Abb. 3.3. Das Start-Arbeitsblatt enthält eine Schaltfläche, um alle erzeugten Datenblätter wieder zu löschen.

Auf dem Start-Arbeitsblatt vereinfachen Hyperlinks die Navigation zu jedem der Daten-Arbeitsblätter. Von jedem Daten-Arbeitsblatt führt ein Hyperlink zurück zum Start-Arbeitsblatt, siehe Abb. 3.3.

Die Demo-Anwendung besteht aus den vier Prozeduren Main, BlaetterEinrichten, BlaetterVerlinken und Aufraeumen. Zwei Konstanten speichern die Namen des Arbeitsblatts mit der Datenliste („sportgruppe“) und des Start-Arbeitsblatts („Start“) zentral und damit leicht änderbar. Zur Veranschaulichung zeigt Abb. 3.4

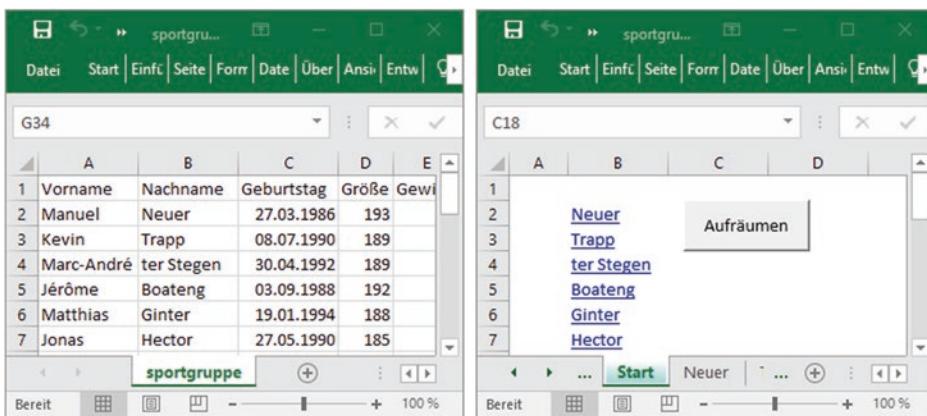


Abb. 3.2 Excel-Arbeitsblätter der Demo-Anwendung „Verlinkte Datenblätter“

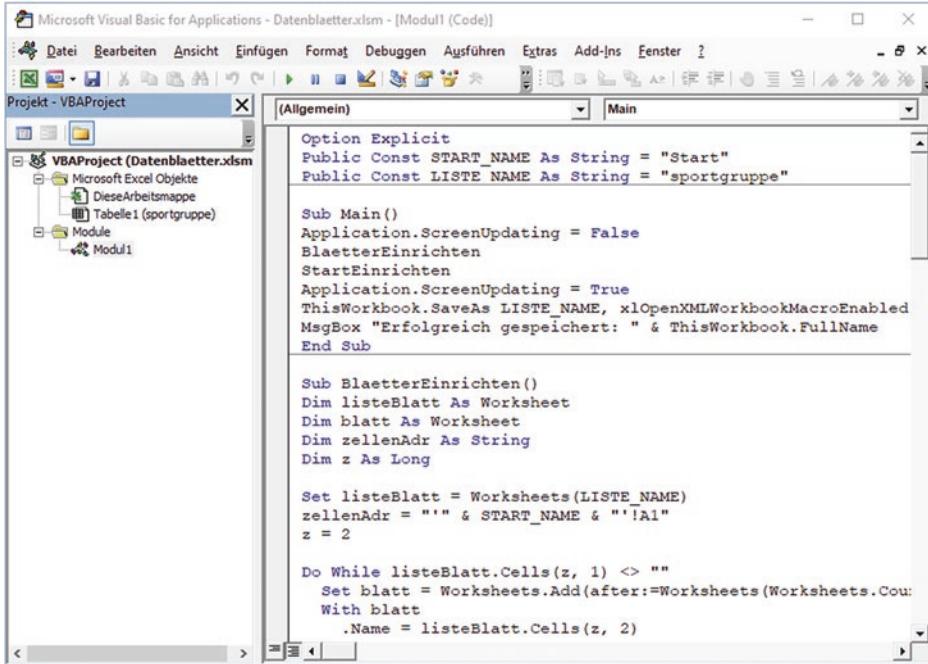


The screenshot shows an Excel spreadsheet titled "sportgruppe_loes.xlsxm - Excel". The active sheet is named "Manuel Neuer". The data is organized in rows 1 through 6:

	A	B	C	D	E	F	G	H	I	J	K
1	Manuel Neuer		1		Start						
2	Geburtstag	27.03.1986									
3	Größe	193									
4	Gewicht	92,00									
5											
6											

The ribbon tabs at the top include: Datei, Start, Einfügen, Seitenlayout, Formeln, Daten, Überprüfen, Ansicht, Entwickertools, Sie wüns..., Anmelden, Freigeben. The status bar at the bottom shows "Bereit" and "100 %".

Abb. 3.3 Durch die Demo-Anwendung „Verlinkte Datenblätter“ erzeugte Arbeitsblätter



The screenshot shows the Microsoft Visual Basic for Applications (VBA) editor. The project tree on the left shows a VBAProject named "Datenblaetter.xlsxm" containing a Microsoft Excel Objekte folder with "DieseArbeitsmappe" and "Tabelle1 (sportgruppe)" sheets, and a Module folder with "Modul1". The code in the main module ("Main") is as follows:

```

Option Explicit
Public Const START_NAME As String = "Start"
Public Const LISTE_NAME As String = "sportgruppe"

Sub Main()
    Application.ScreenUpdating = False
    BlaetterEinrichten
    StartEinrichten
    Application.ScreenUpdating = True
    ThisWorkbook.SaveAs LISTE_NAME, xlOpenXMLWorkbookMacroEnabled
    MsgBox "Erfolgreich gespeichert: " & ThisWorkbook.FullName
End Sub

Sub BlaetterEinrichten()
    Dim listeBlatt As Worksheet
    Dim blatt As Worksheet
    Dim zellenAdr As String
    Dim z As Long

    Set listeBlatt = Worksheets(LISTE_NAME)
    zellenAdr = "" & START_NAME & "!A1"
    z = 2

    Do While listeBlatt.Cells(z, 1) <> ""
        Set blatt = Worksheets.Add(after:=Worksheets(Worksheets.Count))
        With blatt
            .Name = listeBlatt.Cells(z, 2)
        End With
        z = z + 1
    Loop
End Sub

```

Abb. 3.4 Der Code der Demo-Anwendung „Verlinkte Datenblätter“ in der Entwicklungsumgebung

die ersten Code-Zeilen der Demo-Anwendung in der Entwicklungsumgebung. Der gesamte Code ist hier in einem einzigen Modul abgelegt, könnte aber auch gut auf mehrere Module verteilt werden.

3.5.1 Sub Main und Konstantendeklaration

Die Konstantendeklarationen und die Sub Main sind in Abb. 3.4 zu sehen. Die Sub Main dient dazu, die beiden Subs BlaetterEinrichten und StartEinrichten in der richtigen Reihenfolge auszuführen. Vorher schaltet sie die Bildschirmaktualisierung von Excel aus und nachher wieder an.

3.5.1.1 Deaktivieren der Bildschirmanzeige

Das Application-Objekt, das die Sub Main verwendet, steht für die gesamte automatisierte Anwendung, hier Excel. Wenn seine Eigenschaft Application.ScreenUpdating auf True steht, aktualisiert sich die Bildschirmanzeige der Excel-Fenster nach jeder Aktion des VBA-Codes, so wie sie sich auch nach einer manuellen Benutzeraktion aktualisieren würde. Dieses übliche, voreingestellte Verhalten ist wichtig, wenn Benutzer Excel manuell bedienen, weil es ihnen rückmeldet, dass ihre Aktionen erfolgreich ausgeführt wurden. Wenn jedoch ein Skript Aktionen automatisch und sehr schnell ausführt, nehmen Benutzer sie meist nur als Zucken und Flimmern wahr. Trotzdem kostet die Aktualisierung der Bildschirmanzeige Zeit und macht das Skript langsam. Deshalb schalten viele Excel-VBA-Skripte die Aktualisierung der Bildschirmanzeige vorübergehend aus.

Eine ähnliche Eigenschaft des Application-Objekts ist Application.DisplayAlerts. Auch sie steht standardmäßig auf True. Wenn man sie auf False setzt, unterbleiben die Sicherheitsabfragen der Anwendung, zum Beispiel beim Löschen von Arbeitsblättern. Dies nutzt die Sub Aufraeumen, deren Code weiter unten vorgestellt wird.

3.5.1.2 Speichern der Arbeitsmappe

Die Prozedur Main zeigt, wie sich eine Excel-Arbeitsmappe per VBA speichern lässt. Der Objekttyp Workbook bietet dafür neben einer Methode Save auch die hier genutzte Methode SaveAs. Die Demo-Anwendung speichert die Arbeitsmappe mit dem Wert der Konstanten LISTE_NAME als Dateinamen und als Arbeitsmappe mit Makros, also mit der Dateiendung „.xlsm“. Anschließend zeigt sie den Dateinamen mit dem vollständigen Pfad zum Speicherort der Arbeitsmappe in einer MessageBox an. Diese Informationen findet sie im Feld Fullname des Objekts ThisWorkbook. Mit ThisWorkbook lässt sich die Arbeitsmappe ansprechen, die das VBA-Projekt enthält.

3.5.2 Sub BlaetterEinrichten

Die Sub BlaetterEinrichten ist sehr einfach aufgebaut. Sie durchläuft die Zeilen des Listen-Arbeitsblatts aus Abb. 3.2 links mit einer DoLoop-Schleife und führt bei jedem Schleifendurchlauf die gleiche Sequenz von Befehlen aus. Die Schleife läuft, solange die ersten Zellen der Zeilen einen Wert enthalten, der sich vom Leerstring ""

unterscheidet. Danach stoppt sie. Die Befehlssequenz im Schleifenrumpf erzeugt und benennt ein weiteres Daten-Arbeitsblatt wie in Abb. 3.3, füllt dessen Zellen mit Werten und fügt Fomatierungen und einen Hyperlink ein.

```
Sub BlaetterEinrichten()
Dim listeBlatt As Worksheet
Dim blatt As Worksheet
Dim zellenAdr As String
Dim z As Long

Set listeBlatt = Worksheets(LISTE_NAME)
zellenAdr = "" & START_NAME & "!" & "A1"
z = 2

Do While listeBlatt.Cells(z, 1) <> ""
    Set blatt = Worksheets.Add(after:=Worksheets(Worksheets.Count))
    With blatt
        .Name = listeBlatt.Cells(z, 2)
        ' Werte in Zellen schreiben
        .Cells(1, 1) = listeBlatt.Cells(z, 1) & " "
        & listeBlatt.Cells(z, 2)
        .Cells(1, 3) = listeBlatt.Cells(z, 6)
        .Cells(2, 1) = "Geburtstag"
        .Cells(2, 2) = listeBlatt.Cells(z, 3)
        .Cells(3, 1) = "Größe"
        .Cells(3, 2) = listeBlatt.Cells(z, 4)
        .Cells(4, 1) = "Gewicht"
        .Cells(4, 2) = listeBlatt.Cells(z, 5)
        ' Zellen formatieren
        .Cells(1, 1).Style = "Überschrift 3"
        .Cells(1, 3).Interior.Color = RGB(100, 250, 100)
        .Cells(1, 3).Font.Name = "Courier New"
        .Cells(1, 3).Font.Size = 16
        .Cells(1, 3).Font.Bold = True
        .Cells(1, 3).Borders(xlEdgeBottom).Color = RGB(200, 50, 50)
        .Cells(1, 3).Borders(xlEdgeBottom).Weight = xlMedium
        .Cells(2, 2).NumberFormat = "dd.mm.yyyy"
        .Cells(4, 2).NumberFormat = "#0.00"

        .Hyperlinks.Add Anchor:=.Cells(1, 5), _
        Address:"", subaddress:=zellenAdr, _
        ScreenTip:="Gehe zu Start", _
        TextToDisplay:=START_NAME
    End With
    z = z + 1
End Do
```

```

.Columns.AutoFit
End With
z = z +1
Loop
End Sub

```

Noch vor der Schleife weist die Sub das Listen-Arbeitsblatt einer Objekt-Variablen listeBlatt zu, sodass bequem darauf zugegriffen werden kann. Der String zellenAdr wird für die Erzeugung der Hyperlinks zurück zum Start-Arbeitsblatt benötigt. Er bildet das Verweisziel subaddress der Hyperlinks und ist für alle Arbeitsblätter identisch. Daher ist es effizient, ihn einmalig außerhalb der Schleife zu erzeugen. Im Folgenden werden nun interessante Details der Sub erklärt.

3.5.2.1 Einfügen eines Arbeitsblatts an letzter Stelle

Der Befehl `Set blatt = Worksheets.Add(after: = Worksheets(Worksheets.Count))` fügt ein neues Arbeitsblatt nach dem bisher letzten Arbeitsblatt ein und weist es einer Variablen zu. Der Index des bisher letzten Arbeitsblatts entspricht der Anzahl der bisher vorhandenen Arbeitsblätter, also dem Wert `Worksheets.Count`. So wird ein neues Arbeitsblatt am Ende einer Collection `Worksheets` eingefügt.

3.5.2.2 Zellen formatieren

In jedem Schleifendurchlauf schreibt der Code fixe Strings oder Werte aus dem Listen-Arbeitsblatt auf das neu erstellte Arbeitsblatt. Die betreffenden Befehle sind leicht zu verstehen. Die darauffolgenden Befehle greifen auf einige Eigenschaften eines Range-Objekts zu, die seine Formatierung betreffen. Diese Eigenschaften sind unterschiedlich komplex: Die Eigenschaften `Style` und `NumberFormat` speichern einfache Werte. Die Eigenschaften `Interior`, `Font` und `Borders` des Range-Objekts speichern dagegen Objekte. `Borders` ist eine Collection mit vier Objekten des Typs `Border`, so dass ein VBA-Skript jede der vier Rahmenlinien um eine Zelle oder einen Zellenbereich unterschiedlich färben und formatieren kann. Bezeichner wie `xlEdgeBottom` und `xlMedium` sind Konstanten, die für Zahlencodes stehen. An den Zeichen `xl` ist zu erkennen, dass diese Konstanten im Objektmodell von Excel definiert sind. Das heißt, dass sie in VBA ohne Excel nicht verfügbar sind. Die Wirkung der Formatierungsbefehle ist in Abb. 3.3 zu erkennen.

3.5.2.3 Farben definieren

`RGB` ist eine VBA-Funktion, die aus den drei Argumenten `red`, `green` und `blue` einen Long-Zahlencode berechnet, der für einen Farbton steht. Die Parameter `red`, `green` und `blue` geben Farbanteile als Zahlen zwischen 0 und 255 an. Je höher ein Wert ist, desto stärker trägt die betreffende Farbe zum Farbton bei. Wenn alle drei Parameter den maximalen Wert 255 haben, ergibt sich die Farbe Weiß.

3.5.2.4 Hyperlinks einfügen

Wie der Code zeigt, besitzt ein Excel-Arbeitsblatt eine Collection `Hyperlinks`. Die Methode `Hyperlinks.Add` erzeugt einen neuen Link. Die Demo-Anwendung verankert die Hyperlinks in Arbeitsblatt-Zellen, doch auch Zeichenobjekte oder Bilder könnten als Anker dienen. Der Parameter `Anchor` gibt diese Zelle an. Der Wert des Parameters `TextToDisplay` erscheint als klickbarer Link in der Anker-Zelle. Die Parameter `Address` und `SubAddress` definieren das Verweisziel des Links. Der Parameter `Address` könnte ein Verweisziel außerhalb der Arbeitsmappe angeben, zum Beispiel eine Datei im lokalen Verzeichnissystem, eine Webseite oder auch eine E-Mail-Adresse. Hier bleibt dieser Parameter leer, weil der Link die Arbeitsmappe nicht verlässt. Der Parameter `SubAddress` ist optional. Er gibt hier das Arbeitsblatt und die Zelle an, die das Verweisziel des Hyperlinks bilden, konkret „Start!A1“.

Der Befehl `Worksheet.Columns.AutoFit` passt die Breite aller belegten Spalten des Arbeitsblatts an die Breite ihrer Inhalte an.

3.5.3 Sub StartEinrichten

Die Sub `StartEinrichten` erstellt das Start-Blatt. Es wird an zweiter Position und damit nach dem Listen-Blatt eingefügt. Danach durchläuft die Sub mit einer `For-Each`-Schleife die Arbeitsblätter der Arbeitsmappe und erzeugt Hyperlinks, die auf diese Arbeitsblätter verweisen. Das Vorgehen ist dabei gleich wie im vorhergehenden Abschnitt, mit zwei kleinen Unterschieden: Alle Hyperlinks werden auf dem Start-Blatt verankert. Die Verweisziele dieser Hyperlinks werden innerhalb der Schleife erstellt, da sie sich voneinander unterscheiden.

Nach der Erzeugung der Hyperlinks bearbeitet und formatiert die Sub das Start-Blatt und platziert darauf eine Schaltfläche, mit der ein weiteres Makro gestartet werden kann. Der Code der Sub wird zunächst komplett gezeigt und anschließend werden interessante Details genauer betrachtet.

```
Sub StartEinrichten()
Dim startBlatt As Worksheet
Dim blatt As Worksheet
Dim zellenAdr As String
Dim zelle As Range
Dim btn As Button

Dim z As Integer
z = 2

Set startBlatt = Worksheets.Add(after:=Worksheets(1))
startBlatt.Name = START_NAME
```

```

        ' Hyperlinks einfügen
For Each blatt In Worksheets
    With blatt
        If .Name <> START_NAME And .Name <> LISTE_NAME Then
            zellenAddr = "" & .Name & "'!A1"
            blatt.Hyperlinks.Add Anchor:=startBlatt.Cells(z, 2), _
                Address:="", _
                subaddress:=zellenAddr, _
                ScreenTip:="Zu " & .Name, _
                TextToDisplay:=.Name
            z = z + 1
        End If
    End With
Next blatt

        ' Arbeitsblatt formatieren
        ' einheitliche Spaltenbreite festlegen
startBlatt.Columns(1).ColumnWidth = 6
        ' Arbeitsblattregister einfärben

With startBlatt.Tab
    .Color = vbCyan           ' oder vbMagenta, vbRed
    .TintAndShade = 0          ' oder -0.5
End With
        ' Schaltfläche einfügen
Set zelle = startBlatt.Cells(2, 3)
With zelle
    Set btn = startBlatt.Buttons.Add(.Left + 10, .Top, 75, 30)
End With
With btn
    .OnAction = "Aufraeumen"
    .Caption = "Aufräumen "
End With
        ' Startblatt aktivieren, Gitterlinien ausschalten
startBlatt.Activate
ActiveWindow.DisplayGridlines = False
End Sub

```

3.5.3.1 Arbeitsblatt formatieren

Die Sub formatiert die Spalten und das Register des Start-Arbeitsblatts. Ein Befehl greift per Index auf eine einzelne Spalte in der Collection Columns des Arbeitsblatts zu und legt in der Eigenschaft ColumnWidth die Spaltenbreite fest. Weitere Befehle färben das Arbeitsblattregister des Start-Arbeitsblatts wie in Abb. 3.3. Das Arbeitsblattregister ist ein Objekt des Typs Tab. Es ist in der Eigenschaft Tab des Worksheet-Objekts

hinterlegt. Der Eigenschaft Tab.Color wird die vordefinierte Farbcode-Konstante vbCyan zugewiesen. Das Präfix vb zeigt, dass diese Konstante zur Sprache VBA gehört. Die Eigenschaft Tab.TintAndShade des Tab-Objekts beeinflusst die Helligkeit des Farbtöns. Sie kann Werte zwischen –1 und 1 annehmen und damit den Farbton zwischen Schwarz und Weiß abtönen.

3.5.3.2 Einfügen einer Schaltfläche

Die darauf folgenden Befehlszeilen zeigen, wie einfach es ist, mit VBA eine Schaltfläche zu erstellen und mit einem Makro zu verknüpfen. Schaltflächen sind Objekte des Typs Button. Ein Worksheet-Objekt verwaltet Schaltflächen in seiner Collection Buttons. Deren Methode Buttons.Add generiert eine neue Schaltfläche. Sie benötigt die Parameter Top, Left, Width und Height, die die Position der Schaltfläche auf dem Arbeitsblatt und ihre Größe spezifizieren. Die Sub StartEinrichten richtet die Schaltfläche an einer Zelle des Arbeitsblatts aus, indem sie die Koordinaten der linken oberen Ecke der Zelle auch als Koordinaten der Schaltfläche verwendet, mit einem kleinen Offset nach rechts. Wenn das Button-Objekt erstellt ist, nehmen seine Eigenschaften Button.OnAction und Button.Caption den Bezeichner eines Makros beziehungsweise die Beschriftung der Schaltfläche auf. Dadurch ist die Schaltfläche mit dem Makro verknüpft und einsatzfähig. Das Makro Aufraeumen, das hier mit der Schaltfläche verknüpft wird, wird im Abschn. 3.5.4 beschrieben.

3.5.3.3 Arbeitsblatt aktivieren

Interessant sind auch die beiden letzten Befehle dieser Sub. Die Methode Activate des Worksheet-Objekts bringt das Start-Arbeitsblatt in den Vordergrund. Dadurch wird zugleich auch das Excel-Fenster aktiviert, das dieses Arbeitsblatt anzeigt. VBA-Befehle können es als ActiveWindow ansprechen.

3.5.3.4 Gitternetzlinien ausblenden

ActiveWindow ist eine Eigenschaft des Application-Objekts. Darin speichert Excel, welches seiner Fenster gerade aktiv ist. Ähnliche Eigenschaften des Application-Objekts sind ActiveWorkbook, ActiveSheet und ActiveCell. Die Eigenschaft ActiveWindow verweist auf ein Objekt des Objekttyps Window. Für eine gute Optik wird hier seine Eigenschaft DisplayGridLines auf False gesetzt.

3.5.4 Sub Aufraeumen

Die Sub Aufraeumen zeigt, wie Arbeitsblätter gelöscht werden können. Sie läuft mit einer ForEach-Schleife durch die Arbeitsblätter und ruft die Delete-Methode jedes Blatts auf. Lediglich das ursprünglich vorhandene Listen-Arbeitsblatt bleibt erhalten. Die Sub erkennt es an seinem Namen, der in der Konstanten LISTE_NAME hinterlegt ist.

```
Sub Aufraeumen()
Dim blatt As Worksheet
Application.DisplayAlerts = False
For Each blatt In Worksheets
    If blatt.Name <> LISTE_NAME Then
        blatt.Delete
    End If
Next blatt
Application.DisplayAlerts = True
End Sub
```

Die Sub Aufraeumen setzt die Eigenschaft DisplayAlerts des Applications-Objekts vorübergehend auf False. Sonst würde Excel bei jedem Löschen eines Arbeitsblatts warnen und eine DialogBox anzeigen, in der das Löschen bestätigt werden muss.

3.6 Fazit und Ausblick

VBA ist als objektbasierte Programmiersprache darauf ausgelegt, mit Objekten umzugehen und ihre Eigenschaften und Methoden zu nutzen. Mit dem objektbasierten Programmieransatz lassen sich die Komponenten und Funktionen einer Anwendungssoftware wie Excel auf sehr natürliche Weise ansprechen und ansteuern.

Excel eignet sehr gut, um darauf eigene kleine Anwendungen und Tools aufzubauen. Excel-Arbeitsblätter lassen sich für die Anzeige und Eingabe von Daten nutzen und können Bedienelemente wie Schaltflächen, Hyperlinks etc. aufnehmen. Damit lässt sich mit wenig Aufwand eine Bedienoberfläche gestalten. Auch eine einfache Möglichkeit, Daten strukturiert und dauerhaft zu speichern, bringt Excel als Grundlage einer kleinen Digitalisierungslösung bereits mit.



Basis-Techniken

4

Inhaltsverzeichnis

4.1	Fehlerbehandlung	62
4.1.1	Nice to know: Arten von Fehlern	62
4.1.2	Laufzeitfehler abfangen mit On Error	64
4.1.3	Der Befehl On Error GoTo Sprungmarke	65
4.1.4	Das Fehlerobjekt Err	66
4.1.5	Der Befehl On Error Resume Next	67
4.1.6	Umschalten zwischen Fehlerabfangmethoden	68
4.1.7	Nice to know: Kunst und Handwerk der Softwareentwicklung	70
4.2	Objektbibliotheken einbinden und nutzen	71
4.2.1	Word aus Excel-VBA starten und ansteuern	71
4.2.1.1	Die Word-Objektbibliothek in Excel-VBA einbinden	71
4.2.1.2	Neue Instanz von Word starten und Early Binding	72
4.2.1.3	Laufende Instanz von Word ansprechen und Late Binding	73
4.2.1.4	Nice to know: Late Binding versus Early Binding	74
4.2.2	Nice to know: Objektbibliotheken	75
4.2.2.1	Dynamic Link Libraries	75
4.2.2.2	Type Libraries und Object Libraries	76
4.2.3	VBA-Funktionen für den Umgang mit Objektbibliotheken	76
4.2.3.1	Referenzierte Bibliotheken auflisten	76
4.2.3.2	Bibliothek aus VBA referenzieren	77
4.2.3.3	Elemente aus Bibliotheken eindeutig identifizieren	78
4.3	Fazit und Ausblick	78
	Literatur	78

Visual Basic for Applications wird zusammen mit Anwendungssystemen wie Excel, Word, usw. eingesetzt. Jede dieser Anwendungen bringt nicht nur ihr eigenes Objektmodell mit, sie verlangt auch spezifische Herangehensweisen und Programmiermuster.

Doch neben dem gemeinsamen Visual Basic-Sprachkern gibt es noch weitere Programm-elemente und Techniken, die dabei fast immer zum Einsatz kommen, unabhängig vom jeweiligen Anwendungssystem. Dazu gehören Routinen (im Sinne von „Befehlssequenzen“) zur Behandlung von Fehlern. Damit befasst sich der erste Teil dieses Kapitels.

Eine zweite anwendungsübergreifende Technik ist das Einbinden von Objektbibliotheken. Wenn zum Beispiel ein VBA-Skript in Excel als Wirtsanwendung entwickelt wird, ist das Excel-Objektmödell automatisch vorhanden. Das VBA-Skript kann auf Excel-Objekte wie Arbeitsmappen, Arbeitsblätter, Zellen und so weiter zugreifen und die Funktionalität von Excel nutzen. Der zweite Teil dieses Kapitels behandelt, wie sich zusätzliche Objektbibliotheken einbinden lassen, um damit die Funktionalität weiterer Software-Anwendungen oder anwendungsunabhängiger Bibliotheken für ein VBA-Skript zu erschließen.

4.1 Fehlerbehandlung

Bei der Arbeit mit einem Softwaretool öffnet sich plötzlich ein Dialogfenster mit einer technisch aussehenden Fehlermeldung. Es gibt keine Möglichkeit, diesen Fehler zu beheben, und das Tool funktioniert erst nach Abbruch und Neustart wieder. Solche Fehler-situationen lassen sich weitgehend vermeiden, indem man bei der Entwicklung des Softwaretools Routinen zur Fehlerbehandlung einbaut.

4.1.1 Nice to know: Arten von Fehlern

Fehler, die während der Nutzung einer Software auftreten, heißen Laufzeitfehler. Weitere Fehlerarten, mit denen man sich bei der Software-Entwicklung befassen muss, sind Syntaxfehler und Kompilierungsfehler. Syntaxfehler erkennt die VBA-Entwicklungsumgebung schon bei der Eingabe des fehlerhaften Codes wie in Abb. 4.1. Kompilierungsfehler treten zum Beispiel auf, wenn der Code versucht, auf eine unbekannte Prozedur oder Funktion zuzugreifen wie etwa in Abb. 4.2.

Bei Laufzeitfehlern lässt sich das Programm starten und läuft möglicherweise auch eine Weile ohne Probleme. Irgendwann erscheint dann aber eine Fehlermeldung. Viele Laufzeitfehler treten bei jeder Programmausführung auf und sind daher schon bei den Tests während der Entwicklung zu erkennen und zu beheben. Einen solchen Fehler zeigt Abb. 4.3, wo der Code versucht, auf die Zeile 0 eines Excel-Arbeitsblatts zuzugreifen. Andere Laufzeitfehler lassen sich dagegen nicht sicher vermeiden. Dies ist immer dann der Fall, wenn ein Programm von Eingaben oder Ressourcen aus der Außenwelt abhängt, die es nicht kontrollieren kann.

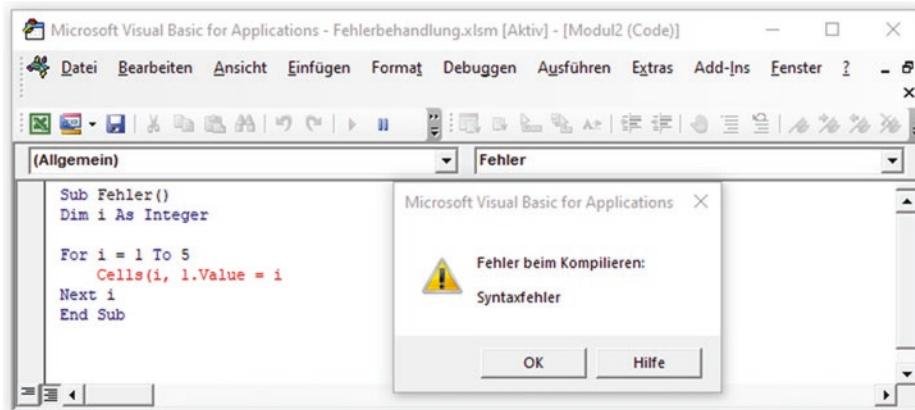


Abb. 4.1 Beispiel Syntaxfehler

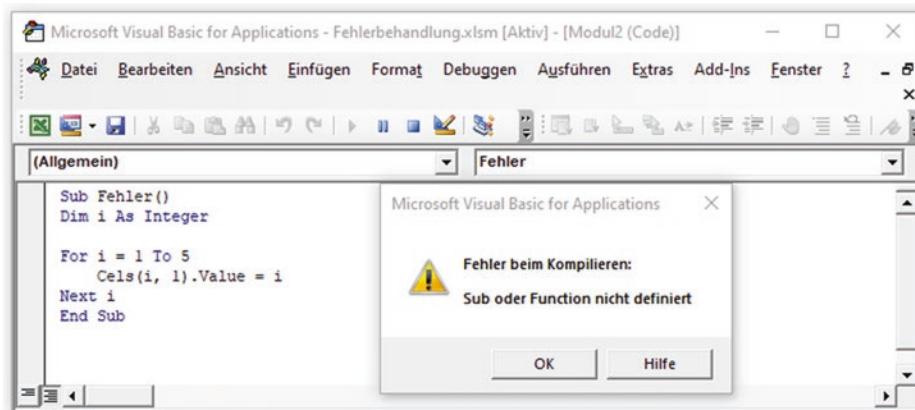


Abb. 4.2 Beispiel Kompilierungsfehler

Typische Beispiele dafür sind:

- Schreiben von Dateien, doch dem Programm fehlt die Schreibberechtigung für den vorgesehenen Dateipfad,
- Einlesen oder Schreiben von Dateien, doch diese existieren nicht (mehr),
- Zugriff auf Mailserver, Datenbanken, Drucker oder ähnliche Ressourcen, die aktuell oder dauerhaft nicht mehr verfügbar sind,
- Einlesen von Daten, die unpassende Formate haben,
- Zugriff auf Objektbibliotheken, die auf dem Computer nicht (mehr) vorhanden sind.

Auch falsche Nutzereingaben sind eine mögliche Ursache für Laufzeitfehler. Um sie zu vermeiden, versucht man die Bedienoberfläche so zu gestalten, dass sie fehlerhafte

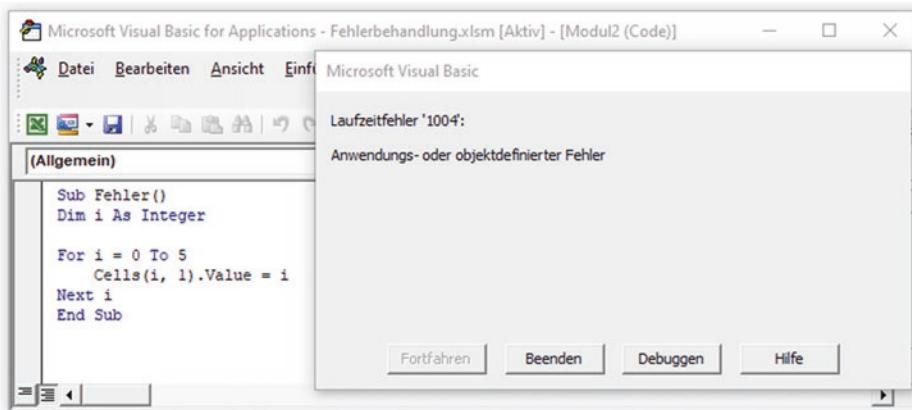


Abb. 4.3 Beispiel Laufzeitfehler

Eingaben gar nicht erst zulässt, zum Beispiel durch Excel-Datenüberprüfung oder Eingabebeschränkungen in UserForms.

In allen diesen Fehlersituationen verlangt die Usability, die Nutzer in verständlicher Weise über Fehlerursachen zu informieren, Abhilfemaßnahmen vorzuschlagen und das Programm sinnvoll weiterarbeiten zu lassen. Zu diesem Zweck bietet VBA die `On Error`-Befehle.

4.1.2 Laufzeitfehler abfangen mit `On Error`

Mit einem `On Error`-Befehl kann man im Programmcode einstellen, wie das Programm weiter ablaufen soll, nachdem ein Laufzeitfehler aufgetreten ist. Ohne einen `On Error`-Befehl lässt das Programm auftretende Laufzeitfehler zum Benutzer durch: es öffnet eine DialogBox mit einer Fehlermeldung und wartet darauf, dass der Benutzer etwas Sinnvolles unternimmt. Mit `On Error`-Befehlen kann man erreichen, dass das Programm einen Laufzeitfehler abfängt und ihn selbst behandelt. VBA bietet folgende `On Error`-Befehle:

- `On Error GoTo Sprungmarke` – springe an eine bestimmte, durch eine Sprungmarke gekennzeichnete Stelle im Programmcode und laufe ab dort weiter
- `On Error Resume Next` – überspringe den fehlerauslösenden Befehl und laufe ab dem darauffolgenden Befehl weiter
- `On Error GoTo 0` – schalte das Fehlerabfangen wieder aus.

Die Sprungmarke kann ein beliebiges Wort oder eine Zahl sein, gefolgt von einem Doppelpunkt. Die 0 in `On Error GoTo 0` bezeichnet jedoch keine Sprungmarke, sondern ist Bestandteil des Befehls.

Der Umgang mit Laufzeitfehlern hat zwei Komponenten: erstens das Abfangen des Fehlers und zweitens das Behandeln des Fehlers. Um beide Aspekte unterscheiden zu können, wird hier die etwas unübliche Formulierung „das Fehlerabfangen“ verwendet. Im Kontext anderer Programmiersprachen sind dafür die englischen Formulierungen „trap an error“ oder „catch an error“ gebräuchlich. In VBA gibt es mit On Error GoTo Sprungmarke und On Error Resume Next zwei Varianten des Fehlerabfangens. Die Programmzeilen, die auf einen abgefangenen Laufzeitfehler reagieren, bezeichnet man als Fehlerbehandlungsroutinen.

On Error-Befehle funktionieren wie Schalter. On Error GoTo Sprungmarke und On Error Resume Next aktivieren die jeweilige Art des Fehlerabfangens. On Error GoTo 0 deaktiviert das Fehlerabfangen. Die Wirkung eines On Error-Befehls beginnt, wenn er ausgeführt wird. Sie erlischt, wenn ein neuer On Error-Befehl ausgeführt wird oder wenn die Sub oder Funktion, die den On Error-Befehl enthält, endet.

4.1.3 Der Befehl On Error GoTo Sprungmarke

Der folgende Demo-Code zeigt, wie On Error Goto Sprungmarke verwendet werden kann. Die Sprungmarke heißt darin Abbruch. Die Prozedur Pruefen soll feststellen, wie viele Zeilen des ersten Arbeitsblatts einer Excel-Arbeitsmappe Werte enthalten. Der Name der Arbeitsmappe ist im Code in einer globalen Variablen hinterlegt. Doch möglicherweise existiert keine Datei dieses Namens am vorgesehenen Dateiverzeichnispfad. Dann fängt die Prozedur den von VBA generierten Fehler ab und verzweigt in einen Codeteil, der eine eigene, informative Fehlermeldung anzeigt. Diese Fehlerbehandlungsroutine beginnt bei der Sprungmarke Abbruch.

```
Const dateiname As String = "daten.xlsx"

Sub Pruefen()
On Error GoTo Abbruch
Dim wbDaten As Workbook

Set wbDaten = Workbooks.Open(ThisWorkbook.Path & "\" & dateiname)
MsgBox (dateiname & " enthält " & _
wbDaten.Worksheets(1).UsedRange.Rows.Count & " Einträge.")
Exit Sub

Abbruch:
MsgBox ("Fehler beim Zugriff auf " & dateiname & "." & _
Chr(13) & "Bitte die Datei hier ablegen: " & ThisWorkbook.Path)
End Sub
```

Der Befehl `Exit Sub` beendet die Prozedur, wenn kein Fehler aufgetreten ist. Ohne diesen Befehl würde die Programmausführung über die Sprungmarke `Abbruch` hinweg weiterlaufen und die Fehlerbehandlungsroutine immer ausführen, nicht nur im Fehlerfall.

Die Wirkung des Befehls `On Error Goto Sprungmarke` endet mit der Prozedur, die den Befehl enthält. Daher ist es hier nicht nötig, sie mit dem Befehl `On Error Goto 0` auszuschalten.

4.1.4 Das Fehlerobjekt `Err`

VBA verwaltet Laufzeitfehler in einem Objekt `Err`. Wenn ein Fehler auftritt, speichert das Objekt `Err` Nummer, Beschreibung und Quelle des Fehlers. VBA-Code kann auf das Objekt `Err` zugreifen und es auslesen. Die Fehlernummer identifiziert den Fehler. Man kann sie nutzen, um auf unterschiedliche Fehler passend zu reagieren. Dies zeigt das folgende Code-Beispiel.

```
Const dateiname As String = "daten.xlsx"
Const blattname As String = "Datum"

Sub Pruefen2()
On Error GoTo Abbruch
Dim wkbDaten As Workbook

Set wkbDaten = Workbooks.Open(ThisWorkbook.Path & "\" & dateiname)
MsgBox (dateiname & " enthält " & _
wkbDaten.Worksheets(blattname).UsedRange.Rows.Count & " Einträge.")
Exit Sub

Abbruch:
    ' während der Entwicklung
MsgBox (Err.Number & " " & Err.Description & " " & Err.Source)
    ' danach nur dieses:
Select Case Err.Number
Case 1004
    MsgBox ("Fehler beim Zugriff auf " & dateiname & "." & _
Chr(13) & "Bitte die Datei hier ablegen: " & ThisWorkbook.Path)
Case 9
    MsgBox ("Fehler in Datei " & dateiname & "." & _
Chr(13) & "Es gibt kein Arbeitsblatt " & blattname & ".")
End Select
End Sub
```

Ein Tipp dazu: Während der Skriptentwicklung ist es von Vorteil, neben eigenen Fehlermeldungen auch Fehlernummer und -beschreibung des Err-Objekts auszugeben wie in diesem Code-Beispiel. So zeigt das Programm auch unerwartete Arten von Fehlern an, während es sie sonst einfach übergeht oder mit falschen, eigenen Fehlermeldungen verschleiert.

Das Fehlerobjekt speichert immer den zuletzt aufgetretenen Fehler. Der Befehl Err.Clear setzt das Fehlerobjekt zurück. Er kommt im folgenden Code-Beispiel zum Einsatz.

4.1.5 Der Befehl On Error Resume Next

Der folgende Demo-Code zeigt ein weiteres Muster der Fehlerbehandlung in VBA. Hier bewirkt On Error Resume Next, dass das Skript zunächst weiter läuft, wenn ein Fehler auftritt. Hinter einem potenziell fehlerauslösenden Befehl prüft es das Fehlerobjekt ab und verzweigt im Fehlerfall in eine Fehlerbehandlungsroutine. Bei der Aktivierung des Fehlerabfangens wird das Fehlerobjekt vorsorglich zurückgesetzt. Am Ende der Fehlerbehandlungsroutinen schaltet der Code mit On Error GoTo 0 das Fehlerabfangen wieder aus. Im folgenden Demo-Code ist dieser Befehl jedoch nicht nötig, weil der Befehl On Error Resume Next ohnehin nur bis zum Prozedurende wirkt.

```
Const dateiname As String = "daten.xlsx"
Const blattname As String = "Datum"

Sub Pruefen3()
Dim wkbDaten As Workbook

On Error Resume Next
Err.Clear

Set wkbDaten = Workbooks.Open(ThisWorkbook.Path & "\\" & dateiname)
If Err.Number <> 0 Then
    MsgBox ("Fehler beim Zugriff auf " & dateiname & "." & _
        Chr(13) & "Bitte die Datei hier ablegen: " & ThisWorkbook.Path)
    Exit Sub
End If

MsgBox (dateiname & " enthält " & _
    wkbDaten.Worksheets(blattname).UsedRange.Rows.Count & " Einträge.")
If Err.Number <> 0 Then
    MsgBox ("Fehler in Datei " & dateiname & "." & _
        Chr(13) & "Es gibt kein Arbeitsblatt " & blattname & ".")
```

```

    Exit Sub
End If

On Error GoTo 0
' hier kann weiterer Code folgen ...
End Sub

```

Dieses Fehlerbehandlungsmuster kommt ohne Sprungmarken aus. Die Fehlerbehandlungsrichtinen stehen nahe beim fehlerauslösenden Code, wodurch der Programmablauf leicht erkennbar ist.

Generell ist es günstig, das Fehlerabfangen nur gezielt für einzelne Code-Abschnitte zu aktivieren und dann wieder abzuschalten. Wenn aus einem Code-Abschnitt mit aktiviertem Fehlerabfangen weitere Prozeduren oder Funktionen aufgerufen werden, in denen ebenfalls Fehlerabfangen aktiviert ist, werden abgefangene Fehler von den aufgerufenen zu den aufrufenden Prozeduren durchgereicht [1]. Das Verhalten des Programms bei Fehlern lässt sich dann nur noch schwer nachvollziehen und durchschauen.

4.1.6 Umschalten zwischen Fehlerabfangmethoden

Zwischen den Fehlerabfangmethoden kann man beliebig umschalten. Der folgende Demo-Code verwendet sowohl On Error GoTo Sprungmarke als auch On Error Resume Next. Die Prozedur InsFolgejahr soll Datumsangaben wie in Abb. 4.4 um ein Jahr aktualisieren. Dabei sind Datenfehler zu erwarten. Die Prozedur behandelt zwei Arten von Laufzeitfehlern auf unterschiedliche Weise: Wenn die zu bearbeitende Datei oder das Datenblatt nicht existieren, gibt sie eine Fehlermeldung aus und endet. Dies erreicht sie mit dem Befehl On Error Goto Abbruch. Wenn der Zugriff auf das Arbeitsblatt erfolgreich war, wechselt die Prozedur zum Fehlerabfangen mit On Error Resume Next. Wenn nun eine Zelle statt des erwarteten Datums einen Text enthält und die Addition deswegen einen Laufzeitfehler verursacht, ignoriert die Prozedur diesen Fehler und arbeitet einfach weiter. Abb. 4.4 zeigt Beispieldaten, wobei links die Daten vor Ausführung der Prozedur zu sehen sind und rechts die Daten danach.

```

Const dateiname As String = "daten.xlsx"
Const blattname As String = "Datum"

Sub InsFolgejahr()
Dim wkbDaten As Workbook
Dim datumspalte As Range
Dim zelle As Range

```

	A	B	C	D
1	14.05.2020	Hans		
2	17.06.2020	Anna		
3	21.07.2020	Gabi		
4	24.08.2020	Franz		
5	Cateen	25.08.2023		
6	31.10.2020	Carina		
7	04.11.2020	Wilhelm		
8	30.12.2020	Gabor		
9	30.12.2020	Lingyan		
10		Kim		
11	01.01.2021	Costa		
12				
13				
14				

	A	B	C	D
1	14.05.2021	Hans		
2	17.06.2021	Anna		
3	21.07.2021	Gabi		
4	24.08.2021	Franz		
5	Cateen	25.08.2023		
6	31.10.2021	Carina		
7	04.11.2021	Wilhelm		
8	30.12.2021	Gabor		
9	30.12.2021	Lingyan		
10		365 Kim		
11	01.01.2022	Costa		
12				
13				
14				

Abb. 4.4 Beispieldaten zur Fehlerbehandlung, links vor und rechts nach Ausführung der Prozedur

```

On Error GoTo Abbruch
Set wkbDaten = Workbooks.Open(ThisWorkbook.Path & "\" & dateiname)
Set datumspalte = wkbDaten.Worksheets(blattname).UsedRange.Columns(1)

On Error Resume Next
For Each zelle In datumspalte.Cells
    zelle.Value = zelle.Value + 365
Next zelle
Exit Sub

Abbruch:
Select Case Err.Number
Case 1004
    MsgBox ("Fehler beim Zugriff auf " & dateiname & "." & _
        Chr(13) & "Bitte die Datei hier ablegen: " & ThisWorkbook.Path)
Case 9
    MsgBox ("Fehler in Datei " & dateiname & "." & _
        Chr(13) & "Es gibt kein Arbeitsblatt " & blattname & ".")
End Select
End Sub

```

4.1.7 Nice to know: Kunst und Handwerk der Softwareentwicklung

Wenn man die Qualität einer Software beurteilt, stehen aus Anwendersicht die funktionalen Anforderungen im Vordergrund: Eine Software muss ihren Zweck erfüllen, sich gut bedienen lassen und dabei zuverlässig funktionieren. Das Software Engineering betrachtet dagegen die Qualität des Codes und damit auch den Programmierstil, denn jede einigermaßen komplexe Software lässt sich auf ganz unterschiedliche Weisen programmieren. Hier zählen dann Kriterien wie Modularität, Verständlichkeit, Kompaktheit, Wartbarkeit, Schnelligkeit der Ausführung, Sparsamkeit beim Speicherbedarf und ähnliches.

Diese Kriterien stehen miteinander im Konflikt: Es ist sicher vorteilhaft, wenn ein Programm mit wenigen Zeilen Code dasselbe leistet wie ein viel umfangreicheres Programm. Besondere Schnelligkeit oder Speichereffizienz braucht aber möglicherweise einige Extrazeilen Code. Ein sehr kompakter Programmierstil kann auch die Verständlichkeit beeinträchtigen. Dies beginnt schon dann, wenn Entwickler so kurze Bezeichner wählen, dass diese den Zweck von Variablen und Prozeduren nicht mehr deutlich machen können.

In manchen Fällen muss Software harte Anforderungen an Schnelligkeit oder Speichersparsamkeit unbedingt erfüllen, etwa bei eingebetteten Echtzeit-Systemen. Auch wenn ein Programm große Datenmengen verarbeiten oder viele Rechenoperationen ausführen muss, ist Effizienz wichtig. Solange jedoch technische und Usability-Anforderungen erfüllt sind, hat im Software Engineering die Verständlichkeit der Software höchste Priorität. Klarer, verständlicher Code erleichtert es, auch die anderen Kriterien einzuhalten und zu optimieren.

Ein Mittel, um die Qualität von Software sicherzustellen, sind sogenannte Entwurfsmuster (pattern). Darunter versteht man Lösungsschablonen für Probleme, die in ähnlicher Form beim Softwareentwurf immer wieder auftreten. Die Demo-Beispiele in den Abschn. 4.1.4 und 4.1.5 zeigen zwei Entwurfsmuster für die Fehlerbehandlung in VBA, eines davon mit Sprungmarke. Sprungmarken werden im Software Engineering sehr kritisch betrachtet, weil sie die Ablauflogik eines Programms schwer durchschau-bar machen. In sehr frühen Programmiersprachen waren Sprünge zu Sprungmarken noch die einzige Möglichkeit, um bestimmte Code-Abschnitte im Programmablauf mehrfach auszuführen oder überhaupt in andere Programmteile zu verzweigen. Die strukturierte Programmierung stellt jetzt mit Funktionen und Prozeduren dafür bessere Mittel zu Verfügung, die die Logik des Programmablaufs klar zeigen. Sprünge und Sprungmarken werden heute in der Programmierung nur noch in ganz bestimmten Situationen eingesetzt. Eine davon ist die Fehlerbehandlung in VBA. Hier ist der oben beschriebene Einsatz von `On Error GoTo` ein anerkanntes und bewährtes Entwurfsmuster.

4.2 Objektbibliotheken einbinden und nutzen

Ein bedeutender Vorteil von VBA ist, dass es mehrere VBA-fähige Anwendungen ansprechen und zusammenarbeiten lassen kann. Dazu bindet man Objektbibliotheken der beteiligten Anwendungen ein. Auch Objektbibliotheken, die nicht zu einer eigenständigen Anwendung gehören, lassen sich einbinden und nutzen, zum Beispiel die Microsoft Scripting Runtime, die im nächsten Kapitel vorgestellt wird. Solange der entwickelte VBA-Code nicht auf viele unterschiedliche Rechnerplattformen portierbar sein muss, ist das Einbinden von zusätzlichen Objektbibliotheken einfach. Der folgende Abschnitt zeigt dazu Beispiele mit Excel und Word und erläutert Optionen, die dabei bestehen. Zwei weitere Abschnitte geben Hintergrundinformationen und stellen VBA-Methoden zum Umgang mit Objektbibliotheken vor, die im Objekttyp des VBA-Projekts angesiedelt sind.

4.2.1 Word aus Excel-VBA starten und ansteuern

Die folgenden Code-Beispiele laufen in Excel als Wirtsanwendung und nutzen Funktionen von Word. Das Vorgehen funktioniert entsprechend auch für andere Wirtsanwendungen und Objektbibliotheken.

4.2.1.1 Die Word-Objektbibliothek in Excel-VBA einbinden

In der VBA-Entwicklungsumgebung öffnet der Menüpunkt **Extras > Verweise** eine DialogBox mit verfügbaren Objektbibliotheken, siehe Abb. 4.5 und 4.6.

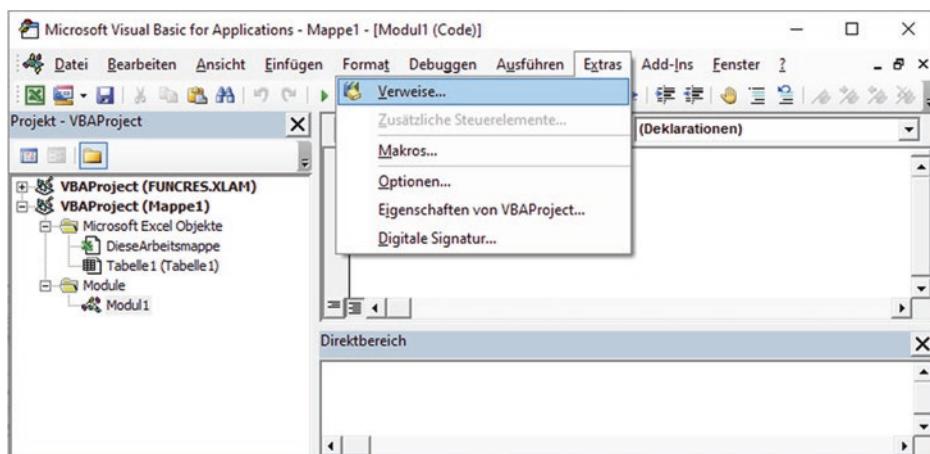


Abb. 4.5 Objektbibliotheken einbinden

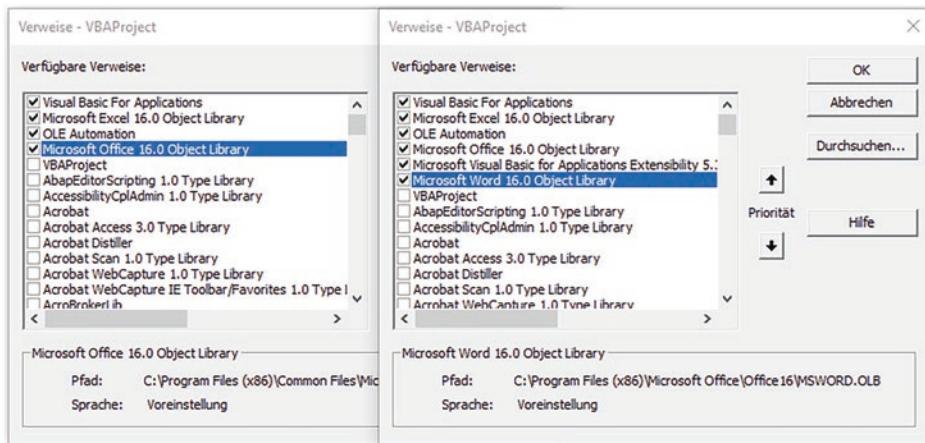


Abb. 4.6 Verfügbare und eingebundene Objektbibliotheken

Bereits eingebundene Objektbibliotheken sind angehakt und stehen oben in der Liste. Wenn die VBA-Entwicklungsumgebung von Excel aus geöffnet wird, sind die Objektbibliothek von Excel, die Office-Objektbibliothek und einige weitere Bibliotheken automatisch eingebunden. Dies ist in Abb. 4.6 links zu sehen. Weitere auf dem Rechner verfügbare Objektbibliotheken erscheinen darunter in alphabetischer Reihenfolge. Welche Objektbibliotheken im Einzelnen verfügbar sind, hängt von der Software ab, die auf dem PC installiert ist. Die Objektbibliothek von Word ist als „Microsoft Word Object Library“ unter M zu finden. Hakt man sie an, zeigt die Verweise-DialogBox sie anschließend ebenfalls oben an, siehe Abb. 4.6 rechts. Ein VBA-Skript kann jetzt neben Excel-Funktionen auch die Funktionalität von Word benutzen.

4.2.1.2 Neue Instanz von Word starten und Early Binding

Um mit Word arbeiten zu können, muss man die Anwendung Word starten. Das folgende Skript zeigt eine Möglichkeit dazu.

```
Sub WordNewEarly()
Dim wordApp As Word.Application
Dim wordDoc As Word.Document

Set wordApp = New Word.Application
On Error GoTo Beenden

' wordApp.Visible = True
' wordApp.Visible = False
Set wordDoc = wordApp.Documents.Add
wordDoc.Content.InsertBefore ("Hallo Welt")
```

```
wordDoc.SaveAs (ThisWorkbook.Path & "\" & "halloWelt")
wordDoc.Close
```

Beenden:

```
wordApp.Quit
Set wordApp = Nothing
End Sub
```

Zunächst wird hier eine Objektvariable für die Anwendung Word deklariert. Diese hat den Typ Application. Da es im Excel-Objektmodell ebenfalls den Typ Application gibt, muss hier der Bibliotheksnname vorangestellt werden, um klarzumachen, welcher der beiden Typen gemeint ist.

Das Schlüsselwort New bewirkt, dass das Skript eine neue Instanz von Word erzeugt und startet. Wird die Eigenschaft Visible auf True gesetzt, öffnet sich die Anwendung Word auf dem Bildschirm. Mit Visible = False läuft sie unsichtbar im Hintergrund. Dies ist effizienter und schneller, weil Bildschirmanzeige und -aktualisierung entfallen. Visible = False ist voreingestellt. Es ist hier lediglich zur Verdeutlichung ebenfalls angegeben.

Das Skript lässt Word ein neues Dokument erzeugen, einen kleinen Text hineinschreiben, es speichern und schließen. Abschließend beendet das Skript die gestartete Word-Instanz und gibt den Speicherplatz frei, indem es die Objektvariable auf Nothing setzt.

Wichtig ist hier, die Applikation durch eine Fehlerbehandlungsroutine auch dann zu beenden, wenn ein Laufzeitfehler auftritt, etwa wenn das Skript das Dokument nicht speichern kann. Sonst läuft die Anwendung unsichtbar weiter und muss manuell mit dem Task Manager oder durch einen PC-Neustart beendet werden.

4.2.1.3 Laufende Instanz von Word ansprechen und Late Binding

Der folgende Beispielcode zeigt eine weitere Möglichkeit, aus Excel-VBA mit der Anwendung Word zu arbeiten. Dieser Code versucht, auf eine laufende Instanz der Word-Application zuzugreifen. Wenn das gelingt, arbeitet er mit dieser Instanz weiter. Wenn aktuell kein Word läuft, löst der Zugriffsversuch einen Fehler aus. Der Code startet dann eine neue Word-Instanz. Wenn bereits ein Word-Dokument offen ist, arbeitet der Code damit. Sonst erzeugt er ein neues Dokument.

```
Sub GetOrStartWordLate()
Dim wordApp As Object      '← Late Binding
Dim wordDoc As Object      '←

'versuche, ein laufendes Word zu nutzen
On Error Resume Next
Set wordApp = GetObject(, "Word.Application")
```

```
If Err.Number <> 0 Then
    ' lösche den Fehler, erzeuge ein neues Word
    Err.Clear
    Set wordApp = CreateObject("Word.Application")
End If

wordApp.Visible = True

If wordApp.Documents.Count < 1 Then
    Set wordDoc = wordApp.Documents.Add
Else
    Set wordDoc = wordApp.Documents(1)
End If
WordDoc.Content.InsertAfter ("Hallo")
wordDoc.Save
End Sub
```

Dieser Code-Schnipsel arbeitet mit Late Binding, während das Skript aus Abschn. 4.2.1.2 Early Binding einsetzt. Sowohl GetObject als auch Dim As New kann man mit Early Binding oder mit Late Binding einsetzen. Den Unterschied zwischen den beiden Binding-Arten erläutert der folgende Abschnitt.

4.2.1.4 Nice to know: Late Binding versus Early Binding

Late Binding versus Early Binding ist ein fortgeschrittenes Thema, das relevant wird, wenn man VBA-Anwendungen entwickelt, die man weiterverbreiten möchte. Der Unterschied zwischen Early Binding und Late Binding liegt in den Deklarationen. Beim Early Binding wird für das Objekt eine Objektvariable des passenden Typs deklariert. Beim Late Binding wird dagegen eine Variable des unspezifischen Typs Object deklariert [2]. Im Falle des Late Binding ist damit während der Skriptentwicklung nicht spezifiziert, welcher Objekttyp hier erwartet wird. Dies kann dann ein Vorteil sein, wenn die entwickelte VBA-Anwendung in unterschiedlichen Systemumgebungen laufen soll, in denen die benötigten Bibliotheken in unterschiedlichen Versionen installiert sind.

Wenn auf einem Rechner eine inkompatible Version einer referenzierten Objektbibliothek installiert ist, schlägt mit Early Binding die Typprüfung fehl und löst einen Fehler aus. Mit Late Binding besteht die Chance, dass das Programm trotzdem funktioniert, wenn nämlich der aufrufende Code nur solche Teile der Objektbibliothek nutzt, die in der installierten Version vorhanden sind. Der Vorteil von Early Binding ist, dass damit bei der Skriptentwicklung automatische Codevervollständigung und Typprüfungen möglich sind.

4.2.2 Nice to know: Objektbibliotheken

Aus technischer Sicht sind die Objektbibliotheken Dynamic Link Libraries (Laufzeitmodule, DLLs). Sie machen die Funktionalität einer Software in Form von Objekten verfügbar. Neu entwickelte Anwendungen, die eine solche Funktionalität benötigen, können die Softwarekomponenten einbinden und verwenden.

4.2.2.1 Dynamic Link Libraries

Die Bezeichnung Dynamic Link kommt daher, dass die nutzende Anwendung auf eine Softwarekomponente der DLL erst dann zugreifen kann, wenn diese Softwarekomponente gestartet oder ausgeführt wird. Man bezeichnet dies als load-time dynamic linking beziehungsweise run-time dynamic linking. Der nicht-dynamische Gegensatz dazu wäre, Softwarekomponenten schon dann in die nutzende Anwendung einzubinden, wenn aus dem Quellcode dieser Anwendung und ebendieser Softwarekomponente eine ausführbare Datei generiert wird.

Dynamic Link Libraries sind eine Spezialität des Betriebssystems Windows. Einige DLLs sind im Betriebssystem enthalten. Weitere DLLs stammen aus Anwendungen, die unter Windows installiert werden. Auch eigenentwickelten Code kann man in DLLs bereitstellen. Wenn die Entwickler die DLLs entsprechend vorbereitet haben, werden diese bei ihrer Installation vom Betriebssystem registriert und sind dann für andere Software sichtbar. Man findet sie dann in der VBA-Entwicklungsumgebung unter **Extras > Verweise** (Englisch: **Tools > References**) und kann sie dort aktivieren, um sie in das eigene Projekt einzubinden, wie in Abb. 4.6. Wie dort auch zu sehen ist, zeigt die DialogBox den Namen und Pfad der gerade markierten DLL-Datei. Typische Dateiendungen von DLLs sind zum Beispiel „.exe“, „.drv“ oder „.dll“. Über die DialogBox kann man auch nicht registrierte und daher nicht gelistete Bibliotheksdateien zum Projekt hinzufügen, wie zum Beispiel in Kap. 11.

Die Verwendung von Dynamic Link Libraries bringt Vorteile: Wenn mehrere Anwendungen dieselbe Funktionalität benötigen, können sie DLLs, die diese Funktionalität bieten, gemeinsam nutzen. Der Code der DLLs muss nur einmal entwickelt und nur einmal gewartet werden und die DLLs benötigt nur einmal Speicherplatz, sowohl auf der Festplatte als auch im Hauptspeicher des Computers, wenn das Programm ausgeführt wird. Ein weiterer Vorteil ist, dass DLLs aktualisiert werden können, ohne die Softwaresysteme, von denen sie genutzt werden, neu installieren zu müssen. Und nicht zuletzt bilden DLLs auch einen Ansatz, um verschiedene Softwaresysteme zu integrieren, das heißt, zusammenarbeiten zu lassen.

Softwaresysteme, die DLLs benutzen, sind davon abhängig, dass diese DLLs in den richtigen Versionen auf dem Computer vorhanden sind. In frühen Windows-Versionen haben die Abhängigkeiten von Dynamic Link Libraries oft zu Problemen geführt, wenn von Softwaresystemen benötigte DLLs von anderen Systemen mit anderen DLL-Versionen überschrieben wurden. Für neuere Windows-Versionen wurde das Prinzip verbessert, sodass Probleme nun wesentlich seltener sind.

4.2.2.2 Type Libraries und Object Libraries

Unter den Verweisen, die die DialogBox der VBA-Entwicklungsumgebung auflistet, finden sich Type Libraries und Object Libraries. Object Libraries enthalten ausführbare Objekte. Type Libraries enthalten Typ-Informationen über Objekte in DLLs. Sie können in eigenen Dateien oder innerhalb der Object Libraries bereitgestellt sein.

Der Zweck von Type Libraries ist, Entwickler bei der Verwendung der DLLs zu unterstützen. Sie beschreiben in expliziter, standardisierter Form die Funktionalität von DLLs in Form von Objekten: welche Objektklassen eine DLL zur Verfügung stellt, welche Elemente diese Objekte haben, mit welchen Parametern man ihre Methoden aufruft usw. Die VBA-Entwicklungsumgebung kann die in einer Type Library beschriebenen Objekte im Objektkatalog anzeigen, sodass man sie dort inspizieren kann. Bei der Programmierung kann die Entwicklungsumgebung mit automatischer Codevervollständigung unterstützen und prüfen, ob der entwickelte Code die Elemente der Object Library korrekt anspricht. DLLs können auch ohne solche Type-Informationen genutzt werden, nur entfällt dann die Unterstützung und Prüfung durch die Entwicklungsumgebung [3].

4.2.3 VBA-Funktionen für den Umgang mit Objektbibliotheken

Der Objekttyp des VBA-Projekts bietet Funktionen, um mit Objektbibliotheken umzugehen. Die folgenden Beispiele laufen in der Wirtsanwendung Excel.

4.2.3.1 Referenzierte Bibliotheken auflisten

Die in einem VBA-Projekt referenzierten Bibliotheken lassen sich mit dem folgenden Code auflisten.

```
Sub VerweiseAuflisten()
    Dim z As Integer
    With ThisWorkbook.VBProject.References
        For z = 1 To .Count
            Cells(z, 1) = .Item(z).GUID
            Cells(z, 2) = .Item(z).Description
            Cells(z, 3) = "!" & .Item(z).major & "." & .Item(z).minor
            Cells(z, 4) = .Item(z).FullPath
            Cells(z, 5) = .Item(z).Name
        Next
    End With
End Sub
```

Ein mögliches Ergebnis zeigt Abb. 4.7. Man findet dort unter anderem die Objektbibliothek VBIDE der VBA-Entwicklungsumgebung. Sie definiert die Objekttypen für Verweise. Die Microsoft Scripting Runtime stellt das Kap. 5 näher vor.

	A	B	C	D	E
1	{000204EF-0000-0000-C000-000000000046}	Visual Basic For Applications	4.2	C:\PROGRA~2\COMMON~1\MICROS~1\VBA\BA\VBA7.1\VBE7.DLL	VBA
2	{00020813-0000-0000-C000-000000000046}	Microsoft Excel 16.0 Object Library	1.9	C:\Program Files (x86)\Microsoft Office\Office16\EXCEL.EXE	Excel
3	{00020430-0000-0000-C000-000000000046}	OLE Automation	2.0	C:\Windows\SysWOW64\stdole2.tlb	stdole
4	{2DF8D04C-5BFA-101B-BDE5-00AA0044DE52}	Microsoft Office 16.0 Object Library	2.8	C:\Program Files (x86)\Common Files\Microsoft	Office
5	{00020905-0000-0000-C000-000000000046}	Microsoft Word 16.0 Object Library	8.7	C:\Program Files (x86)\Microsoft Office\Office16\MSWORD.OLB	Word
6	{0002E157-0000-0000-C000-000000000046}	Microsoft Visual Basic for Applications Extensibility 5.3	5.3	C:\Program Files (x86)\Common Files\Microsoft	VBIDE
7	{420B2830-E718-11CF-893D-00A0C9054228}	Microsoft Scripting Runtime	1.0	C:\Windows\SysWOW64\scrrun.dll	Scripting
8					
9					

Abb. 4.7 Beispieldaten zu „Referenzierte Objektbibliotheken auflisten“

4.2.3.2 Bibliothek aus VBA referenzieren

Bibliotheken lassen sich nicht nur manuell, sondern auch mittels Programmcode einbinden. Eine Objektbibliothek lässt sich dabei über ihre DLL-GUID oder ihren Pfad und Dateinamen identifizieren. Die GUID=Global Unique Identifier ist eine eindeutige Nummer, die der Hersteller einer DLL festgelegt hat.

Das folgende Code-Beispiel referenziert die Objektbibliothek Microsoft Scripting Runtime. Zuvor prüft es, ob die Bibliothek bereits referenziert ist. Es nutzt dazu das Objektmodell der VBA-Entwicklungsumgebung, hier Microsoft Visual Basic for Applications Extensibility 5.3 siehe Abb. 4.6.

```
Sub Referenzieren()
Dim verw As VBIDE.Reference
Dim verws As VBIDE.References
Dim IstAktiv As Boolean

IstAktiv = False
Set verws = ThisWorkbook.VBProject.References
For Each verw In verws
    If verw.Name = "Scripting" Then
        IstAktiv = True
        Exit For
    End If
Next verw

If Not IstAktiv Then
    ' verws.AddFromGuid "{420B2830-E718-11CF-893D-00A0C9054228}", 1, 0
    verws.AddFromFile ("C:\Windows\SysWOW64\scrrun.dll")
End If
End Sub
```

4.2.3.3 Elemente aus Bibliotheken eindeutig identifizieren

Wenn mehrere in ein VBA-Projekt eingebundene Objektbibliotheken für Elemente denselben Namen verwenden, kann es zu Namenskonflikten kommen. Ein Beispiel dafür ist `Application`. Sowohl das Excel-Objektmodell als auch das Word-Objektmodell enthalten einen so bezeichneten Objekttyp. Um die Mehrdeutigkeit aufzulösen, gibt man zusätzlich zum Namen des Objekts auch den Namen der Bibliothek an, also zum Beispiel `Excel.Application` oder `Word.Application`.

4.3 Fazit und Ausblick

Beim Einsatz von Software entstehen häufig Situationen, in denen ein Programm Daten, Benutzereingaben oder andere Ressourcen nicht oder nicht in der richtigen Form findet, sodass interne Fehler auftreten. Eine effektive Fehlerbehandlung fängt solche Fehler ab und sorgt dafür, dass das Programm in sinnvoller Weise weiterläuft. Für VBA gibt es bewährte Entwurfsmuster für die Fehlerbehandlung.

Durch die Möglichkeit, Objektbibliotheken einzubinden, wird VBA sehr mächtig, denn damit lässt sich die komplexe und spezialisierte Funktionalität von Word, Excel und weiteren Anwendungssystemen in selbst entwickelten Tools nutzen. Daneben stehen auch Objektbibliotheken zur Verfügung, die nicht zu einer Anwendungssoftware gehören. Eine solche Objektbibliothek ist die Microsoft Scripting Runtime, die das nächste Kapitel behandelt.

Literatur

1. o365devx u. a., „On Error statement (VBA)“, Microsoft Learn Office VBA, 30. März 2022. <https://learn.microsoft.com/en-us/office/vba/language/reference/user-interface-help/on-error-statement> (zugegriffen 25. März 2023).
2. K. Dollard, „Early and Late Binding – Visual Basic“, 15. September 2021. <https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/early-late-binding/> (zugegriffen 15. August 2022).
3. Microsoft, „Contents of a Type Library“, 31. Mai 2018. <https://docs.microsoft.com/en-us/previous-versions/windows/desktop/automat/contents-of-a-type-library> (zugegriffen 12. August 2022).



Dateiverzeichnissystem

5

Inhaltsverzeichnis

5.1	Überblick über das Objektmodell des Dateiverzeichnissystems	80
5.2	Code-Beispiele	80
5.2.1	Auf Laufwerk, Verzeichnis und Datei zugreifen	81
5.2.2	Auf eine Datei zugreifen und Informationen anzeigen	82
5.3	Demo-Anwendung „Dokumente-Massenbearbeitung“	83
5.4	Code-Beispiel „Log-Datei“	85
5.5	Fazit	87

Die Microsoft Scripting Runtime ist eine Objektbibliothek, die nicht zu einer Anwendungssoftware gehört. Sie definiert zwei unabhängige Objekte: die Klasse `Dictionary` und das `FileSystemObject`. Instanzen der Klasse `Dictionary` speichern Datensammlungen als Schlüssel-Wert-Paare und bieten damit eine Alternative zu Arrays und Collections. In diesem Kapitel wird das `FileSystemObject` näher vorgestellt. Es vereinfacht den Umgang mit dem Dateiverzeichnissystem des Rechners. Sein Einsatz lohnt sich für umfangreiche Operationen mit Dateien und Dateiverzeichnissen, etwa beim Durchsuchen oder Organisieren des Dateiverzeichnissystems oder zur Massenbearbeitung von Dateien. Es bietet außerdem Methoden, um Textdateien zu schreiben.

Eine Demo-Anwendung dieses Kapitels nutzt das `FileSystemObject`, um Word-Dateien aus mehreren Dateiverzeichnissen zu sammeln und ihnen ein einheitliches Design zuzuweisen. Ein weiteres Demo-Skript durchläuft ein Dateiverzeichnis und seine Unterdateiverzeichnisse in beliebiger Tiefe. Dieses Skript beruht auf rekursiver Programmierung und ist damit auch ein Beispiel für eine interessante Programmiertechnik. Abschließend zeigt ein Code-Beispiel eine einfache Lösung, um die Aktivitäten einer Anwendung in einer Log-Datei zu protokollieren.

5.1 Überblick über das Objektmodell des Dateiverzeichnissystems

Das Objektmodell in der Microsoft Scripting Runtime bildet die Baumstruktur eines Dateiverzeichnissystems nach. Abb. 5.1 stellt einen Ausschnitt daraus grafisch dar.

Top-Element dieses Objektmodells ist das `FileSystemObject`. Dieses speichert in seinem Feld `Drives` eine Collection von Objekten des Typs `Drive`, die die Laufwerke des jeweiligen Rechners abbilden. Die zentralen Objekttypen sind `Drive`, `Folder` und `File`.

- Ein `Drive`-Objekt hat ein Feld `RootFolder`, das ein Objekt des Typs `Folder` speichert.
- Jedes `Folder`-Objekt hat ein Feld `Subfolders` und ein Feld `Files`.
- Das Feld `Files` speichert eine Collection von Objekten des Typs `File`.
- Das Feld `Subfolders` speichert eine Collection von Objekten des Typs `Folder`.

Da jedes dieser `Folder`-Objekte eine Collection mit weiteren `Folder`-Objekten besitzt, kann prinzipiell eine beliebig tief verschachtelte Struktur entstehen. Praktisch gibt es natürlich eine Grenze. Die folgenden Code-Beispiele demonstrieren den Umgang mit Objekten und Methoden aus diesem Objektmodell.

5.2 Code-Beispiele

Die Microsoft Scripting Runtime bildet kein eigenständiges Anwendungssystem und bringt auch keine Bedienoberfläche mit. Sie wird deshalb aus anderen VBA-fähigen Anwendungen referenziert. Die Code-Beispiele dieses Kapitels verwenden Excel als Wirtsanwendung. Damit die Code-Beispiele funktionieren, muss die Microsoft Scripting

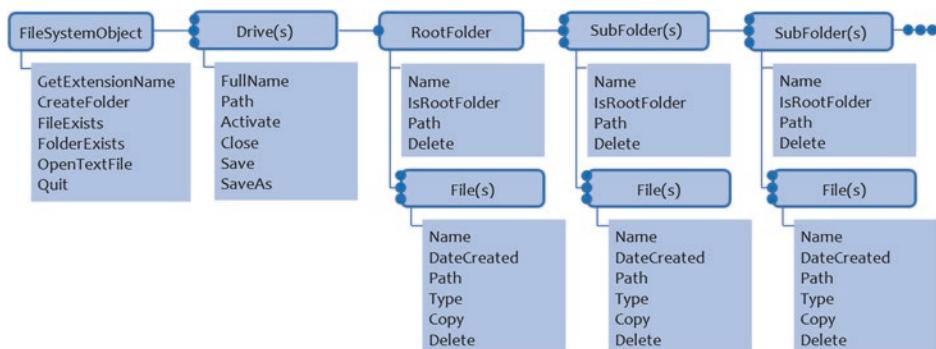


Abb. 5.1 Ausschnitt aus dem Objektmodell der Microsoft Scripting Runtime

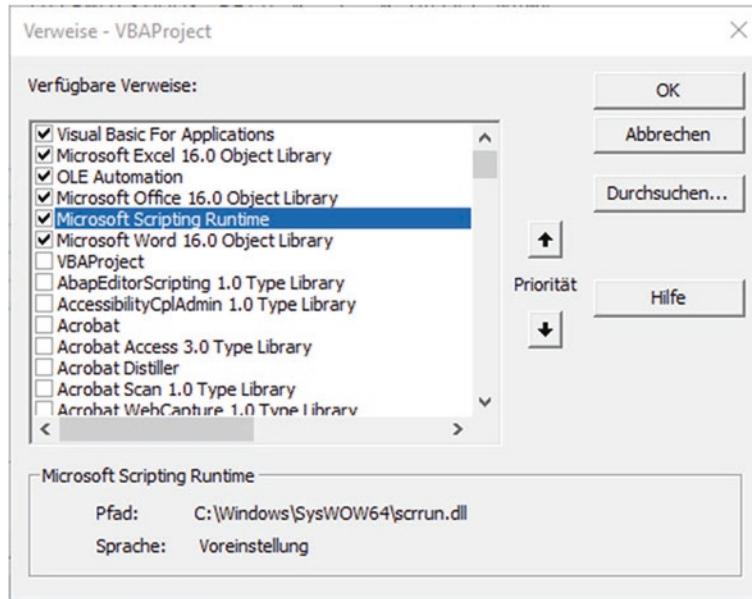


Abb. 5.2 Einbinden der Objektbibliothek Microsoft Scripting Runtime

Runtime wie in Abb. 5.2 in das VBA-Projekt eingebunden sein. Ein Code-Beispiel dieses Kapitels benötigt außerdem noch die Word-Objektbibliothek, weshalb diese in Abb. 5.2 ebenfalls angehakt ist.

5.2.1 Auf Laufwerk, Verzeichnis und Datei zugreifen

Laufwerke, Dateien und Verzeichnisse lassen sich ausschließlich per Namen oder in einer ForEach-Schleife ansprechen. Zugriffe per Index, etwa Drives(1) oder Folders(1) funktionieren nicht. Das folgende Code-Beispiel zeigt einige Befehle zum Zugriff auf Laufwerke, Verzeichnisse und Dateien:

```
Sub FSODemo()
Dim FSO As New FileSystemObject
Dim laufw As Drive
Dim dat As File
Dim verz As Folder

Debug.Print "Aktuelles Verzeichnis: " & FSO.GetFolder(".").Path
Debug.Print "Eigener Pfad: " & ThisWorkbook.Path
Debug.Print
```

```

Set laufw = FSO.Drives("c")
Debug.Print "Wurzelverzeichnis: " & laufw.RootFolder.Path
Debug.Print "*** Anzahl Verzeichnisse: " & _
laufw.RootFolder.SubFolders.Count
For Each verz In laufw.RootFolder.SubFolders
    Debug.Print verz.Name
Next verz

Debug.Print vbLf & "*** Anzahl Dateien: " & _
laufw.RootFolder.Files.Count
For Each dat In laufw.RootFolder.Files
    Debug.Print dat.Name
Next dat
End Sub

```

Die Methode `GetFolder` des `FileSystemObject` erwartet als Parameter einen String, der einen Verzeichnispfad darstellt. Der Punkt in `FSO.GetFolder(".")` ist ein Platzhalter für das jeweils aktuelle Verzeichnis. Der Ausdruck `ThisWorkbook.Path` liefert den Pfad der Excel-Arbeitsmappe, die das VBA-Skript enthält. Beide stimmen nicht immer überein.

Das Skript listet mit zwei Schleifen alle Verzeichnisse und alle Dateien im Wurzelverzeichnis des Laufwerks „c“ auf dem jeweiligen Rechner auf. Es erzeugt Leerzeilen, um die Ausgaben der `Debug.Print`-Befehle zu untergliedern. Dafür verwendet es zwei verschiedene Methoden: erstens einen `Debug.Print`-Befehl ohne Parameter und zweitens das Verketten von `vbLF` mit einem String. `vbLF` ist eine VisualBasic-Konstante für das Zeilenvorschub-Zeichen.

5.2.2 Auf eine Datei zugreifen und Informationen anzeigen

Das folgende Code-Beispiel prüft, ob eine bestimmte Datei vorhanden ist, und gibt Informationen über sie aus. Es zeigt nur eine Auswahl der verfügbaren Datei-Informationen, das `File`-Objekt und das `FileSystemObject` könnten noch weitere Infos liefern. Das Code-Beispiel erwartet, dass eine Datei „Datei.docx“ in dem Verzeichnis vorhanden ist, das auch die Excel-Arbeitsmappe mit diesem VBA-Skript enthält.

```

Sub DateiInfo()
Const DATEI_NAME As String = "Datei.docx"
Dim FSO As New FileSystemObject
Dim datei As File
Dim dateipfad As String

dateipfad = ThisWorkbook.Path & "\" & DATEI_NAME
Debug.Print dateipfad

```

```
If FSO.FileExists(dateipfad) Then
    Set datei = FSO.GetFile(dateipfad)
    With datei
        Debug.Print .Name
        Debug.Print .DateCreated
        Debug.Print .Type
        Debug.Print .Path
        Debug.Print FSO.GetBaseName(.Path)
        Debug.Print FSO.GetExtensionName(.Path)
    End With
Else
    Debug.Print dateipfad & " ist nicht da"
End If
End Sub
```

Wie das Code-Beispiel zeigt, besitzt neben dem File-Objekt auch das FileSystemObject Methoden zum Abruf von Informationen über Dateien wie etwa die Methoden GetExtensionName und GetBaseName. Sie erwarten als Parameter einen String mit dem Pfad einer Datei einschließlich ihres Dateinamens. GetExtensionName liefert die Dateiendung und GetBaseName den Namen der Datei ohne Pfad und ohne Dateiendung. Das ist zum Beispiel nützlich, wenn man Dateien umbenamt oder in einem neuen Format abspeichert.

5.3 Demo-Anwendung „Dokumente-Massenbearbeitung“

Die Demo-Anwendung „Dokumente-Massenbearbeitung“ sammelt Word-Dokumente, die verstreut in mehreren Dateiverzeichnissen liegen, in einem einzigen Dateiverzeichnis und bearbeitet sie. Sie durchsucht dabei ein angegebenes Dateiverzeichnis und alle darunter liegenden Unterdateiverzeichnisse. Um in beliebig tiefe Unterdateiverzeichnisstrukturen absteigen zu können, verwendet die Demo-Anwendung eine rekursive Prozedur. Rekursiv bedeutet, dass die Prozedur sich selbst aufruft (rekursiv=zu sich selbst zurückkehrend).

Die Demo-Anwendung besteht aus zwei Subs. Die Sub DateienSammeln ist die Haupt-Prozedur. Sie legt ein neues Verzeichnis mit dem Titel „Worddateien“ an und stößt die Sub VerzeichnisBearbeiten an. Falls jedoch das Verzeichnis „Worddateien“ schon existiert, bricht die Sub DateienSammeln mit einer Info ab. Die Sub VerzeichnisBearbeiten führt die eigentliche Arbeit aus. Sie ist immer in einem bestimmten Dateiverzeichnis aktiv: zunächst findet sie alle Word-Dokumente, die es enthält, und bearbeitet sie. Dann ermittelt sie die Unterdateiverzeichnisse des Dateiverzeichnisses und wird der Reihe nach in jedem davon auf dieselbe Weise aktiv. Dies bedeutet, dass die Sub VerzeichnisBearbeiten sich mit einem anderen Dateiverzeichnis als Argument selbst wieder aufruft.

Da die Unterverzeichnisse Dateien mit identischen Namen enthalten können, muss verhindert werden, dass diese sich beim Kopieren gegenseitig überschreiben. Deshalb erzeugt die Sub VerzeichnisBearbeiten neue Dateibezeichner. Dazu fügt sie die Bezeichner der bisher durchlaufenen Dateiverzeichnisse einschließlich des aktuellen Dateiverzeichnisses in der Variablen praefix zusammen und stellt dieses praefix dem Namen jeder Datei voran.

```

Const ZIELPFAD As String = "Worddateien"
Const ARBEITSPFAD As String = "FSO_Beispiele"
' Const NEUER_STIL = "Linien (markant)"           ' v2
Dim FSO As New Scripting.FileSystemObject
' Dim WordApp As New Word.Application             ' v2

Sub DateienSammeln()
Dim arbeitsverz As Folder
Dim zielverz As Folder
Dim dat As File
Dim pfade As String

Err.Clear
On Error GoTo Ende
pfad = ThisWorkbook.Path & "\"

If FSO.FolderExists(pfad & ZIELPFAD) Then
    MsgBox ("ZIELPFAD existiert schon. Abbruch.")
Else
    Set arbeitsverz = FSO.GetFolder(pfad & ARBEITSPFAD)
    Set zielverz = FSO.CreateFolder(pfad & ZIELPFAD)
    VerzeichnisBearbeiten zielverz.Path & "\", arbeitsverz
End If

Ende:
If Err.Number <> 0 Then
    Debug.Print Err.Description
End If
' WordApp.Quit SaveChanges:=WdSaveOptions.wdDoNotSaveChanges ' v2
' Set WordApp = Nothing                                     ' v2
End Sub

Sub VerzeichnisBearbeiten(präfix As String, verzeichnis As Folder)
Dim dat As File
' Dim dok As Document                                      ' v2
Dim unterverz As Folder
Dim präfixneu As String
Dim datpfad As String

```

```
praefixneu = praefix & verzeichnis.Name & "_"
For Each dat In verzeichnis.Files
    datpfad = praefixneu & dat.Name
    Debug.Print datpfad
    If dat.Type = "Microsoft Word-Dokument" Then
        dat.Copy (datpfad)                                ' v1
        ' Set dok = WordApp.Documents.Open(dat.Path)      ' v2
        ' dok.ApplyQuickStyleSet2 (NEUER_STIL)            ' v2
        ' dok.SaveAs (datpfad)                            ' v2
        ' dok.Close                                     ' v2
    End If
Next dat
For Each unterverz In verzeichnis.SubFolders
    VerzeichnisBearbeiten praefixneu, unterverz
Next unterverz
End Sub
```

Die Demo-Anwendung „Dokumente-Massenbearbeitung“ kommt in zwei Varianten. Die erste Variante kopiert wie beschrieben alle Word-Dokumente aus einem Dateiverzeichnis und dessen Unterverzeichnissen in das neue Verzeichnis „Worddateien“. Die zweite Variante kopiert die Word-Dokumente nicht nur, sondern weist ihnen auch noch ein neues Design zu. Damit dies funktioniert, muss die Word-Objektbibliothek in das VBA-Projekt eingebunden sein und eine Instanz von Word gestartet werden. Man könnte den Code auch direkt in Word als Wirtsanwendung entwickeln und ausführen statt in Excel. In Excel kann der Code etwas schneller arbeiten, weil Word unsichtbar im Hintergrund laufen kann.

Im Code sind Befehle, die nur in einer der beiden Varianten benötigt werden, mit „v1“ beziehungsweise „v2“ markiert. Die Befehle für „v2“ sind auskommentiert. Damit der Code etwas bewirkt, müssen natürlich passende Dateiverzeichnisse, Word-Dateien und der anzuwendende Word-Stil vorhanden sein.

5.4 Code-Beispiel „Log-Datei“

Das FileSystemObject kann Textdateien erzeugen und in Textdateien schreiben, ohne auf eine weitere Anwendung (wie etwa Microsoft Word) zurückzugreifen. Die Textdateien sind Objekte des Typs TextStream. Dessen Methoden Write, WriteLine und WriteBlankLines schreiben in die Datei. WriteLine(Text) schreibt einen Text und einen Zeilenvorschub, Write(Text) schreibt einen Text ohne anschließenden Zeilenvorschub und WriteBlankLines(n) erzeugt n leere Zeilen. Ein TextStream-Objekt bietet außerdem Methoden, um Textdateien zu lesen und ihren Text als Strings im VBA-Code verwendbar zu machen.

Das folgende Code-Beispiel zeigt eine einfache Lösung, um Aktionen eines Skripts in einer Log-Datei zu protokollieren. Die Prozedur SchreibeLog hängt eine neue Zeile an eine Log-Datei an. Hier enthält diese Zeile den aktuellen Zeitpunkt, den Namen des angemeldeten Nutzers und eine Info. Den Nutzernamen liefert die Systemumgebung (Environ). Ein Aufruf der Prozedur SchreibeLog könnte zum Beispiel in der Demo-Anwendung „Dokumente-Massenbearbeitung“ die Debug.Print-Befehle ersetzen, um den Ablauf der Demo-Anwendung in einer Datei zu protokollieren.

Für jeden Tag, an dem protokolliert wird, legt das Skript eine neue Protokolldatei an. Diese erhält den Namen „yy-mm-dd_log.txt“, wobei „yy-mm-dd“ für das jeweils aktuelle Datum steht. Dazu arbeitet die Sub InitLog mit der Methode OpenTextFile des FileSystemObject. Ihr erster Parameter ist der Name der Textdatei einschließlich Pfad. Als zweiter Parameter ist hier ForAppending angegeben. So wird neuer Text an die Textdatei angehängt statt diese zu überschreiben. Andere Optionen für diesen Parameter wären ForReading oder ForWriting. Der dritte Parameter hat den Wert True, was bewirkt, dass die Textdatei neu erzeugt wird, falls sie noch nicht existiert.

```
Private LogDat As TextStream
Private Const LOGVERZEICHNIS As String = "Logs\"

Sub Main()
    SchreibeLog ("Dies ist ein Text")
    SchreibeLog ("Dies ist noch ein Text")
    CloseLog
End Sub

Sub SchreibeLog(info As String)
If LogDat Is Nothing Then
    InitLog
End If
LogDat.WriteLine (Now() & " - " & Environ("username") & " - " & info)
End Sub

Private Sub InitLog()
Dim FSO As FileSystemObject
Dim logVerz As String
Dim logDatPfad As String

Set FSO = New FileSystemObject
logVerz = ThisWorkbook.Path & "\" & LOGVERZEICHNIS & "\"
logDatPfad = logVerz & "\" & Format(Date, "yy-mm-dd") & "_log.txt"

If Not FSO.FolderExists(logVerz) Then
    FSO.CreateFolder (logVerz)
```

```
End If
Set LogDat = FSO.OpenTextFile(logDatPfad, ForAppending, True)
End Sub

Public Sub CloseLog()
If Not LogDat Is Nothing Then
    LogDat.Close
    Set LogDat = Nothing
End If
End Sub
```

Häufige Dateizugriffe belasten das System und verlangsamen ein Skript. Um dieses Problem zu verbessern, öffnet das Skript die Protokolldatei beim ersten Schreibzugriff und hält sie dann für folgende Schreibzugriffe offen. Die Sub `CloseLog` kann gerufen werden, um eine offene Protokolldatei zu schließen, damit das Betriebssystem sie zum Löschen, Verschieben etc. freigibt.

5.5 Fazit

Das `FileSystemObject` der Microsoft Scripting Runtime bietet komfortable Methoden, um mit Dateiverzeichnissen und Dateien zu arbeiten. Mit kompaktem VBA-Code lassen sich Aufgaben automatisieren, deren manuelle Ausführung viel Aufwand kosten würde, zum Beispiel, das Design vieler Word-Dokumente zu vereinheitlichen. Das Objektmodell des Dateiverzeichnissystems in der Microsoft Scripting Runtime ist rekursiv aufgebaut. Eine rekursive Prozedur eignet sich sehr gut, um es zu durchsuchen.



Word

6

Inhaltsverzeichnis

6.1	Das Objektmodell von Word	90
6.1.1	Die Collection Documents	90
6.1.2	Stories	92
6.1.3	Der Objekttyp Range	92
6.1.4	Text ersetzen, einfügen und löschen	94
6.2	Demo-Anwendung „Word aus Excel“.....	96
6.2.1	Teil „Excel2Word“ der Demo-Anwendung.....	97
6.2.2	Teil „Word2Excel“ der Demo-Anwendung.....	99
6.3	Textmarken und Tabellen.....	102
6.4	Demo-Anwendung „Produktblätter“.....	103
6.5	Dokumente automatisiert bearbeiten mit Suchen und Ersetzen	106
6.5.1	Der Objekttyp Find	106
6.5.2	Der Objekttyp Selection	107
6.5.3	Code-Beispiel: Wörter innerhalb der Auswahl formatieren.....	107
6.5.4	Code-Beispiel: Wörter mit bestimmtem Absatzformat formatieren	109
6.6	Demo-Anwendung „Etiketten-Tool“.....	110
6.6.1	Der Objekttyp DocumentProperty.....	111
6.6.2	Der Objekttyp Field.....	112
6.6.3	Erzeugen von Barcodes	112
6.6.4	Code der Demo-Anwendung	113
6.7	Fazit	115
	Literatur.....	115

Formatierungen vereinheitlichen, Designs austauschen, Produktblätter, Kataloge oder Zeugnisse automatisiert erstellen, Word-Dokumente aus anderen Anwendungen heraus generieren, Dokumente mit Schlagwörtern und Meta-Information versehen und

automatisch ablegen – die Automatisierung von Word eröffnet viele interessante Anwendungsmöglichkeiten. Die Textverarbeitungsfunktionen von Word sind auch in Outlook integriert. Auch deshalb ist es lohnenswert, sich damit zu befassen.

Word-Dokumente sind im Vergleich zu Excel-Arbeitsmappen schwach strukturiert. Anders als die einfach zu adressierenden Arbeitsblätter und Zellen einer Excel-Arbeitsmappe wirkt die Struktur eines Word-Dokuments für VBA-Entwickler zunächst oft unübersichtlich. Das Kapitel startet mit einem Einblick in das Objektmodell von Word. Dann zeigt es verschiedene Ansätze, um Dokumente in unterschiedlichen Anwendungsszenarien automatisch zu erzeugen. Weitere Demos befassen sich mit der Formatierung von Texten. Eine Schlüsseltechnik der Digitalisierung sind Barcodes, denn sie verknüpfen physische Objekte mit der digitalen Welt. Eine Demo-Anwendung zeigt, wie man sie mit Word-Funktionen generieren kann.

Die Code-Beispiele dieses Kapitels demonstrieren darüber hinaus zwei Programmiertechniken, die auch unabhängig vom Word-Objektmodell interessant und vielseitig einsetzbar sind: Wie man eigene, strukturierte Datentypen definiert und wie man mit VBA auf die Dokumenteneigenschaften von Office-Dateien zugreifen kann.

6.1 Das Objektmodell von Word

Das Objektmodell von Word spiegelt den großen Funktionsumfang der Anwendung Word wieder. Es enthält Objekttypen für Bibliographien, Wörterbücher, Textbausteine und vieles mehr. Diese Einführung konzentriert auf den Ausschnitt des Objektmodells, der den Inhalt von Word-Dokumenten betrifft. Ein Teil davon ist in Abb. 6.1 grafisch dargestellt.

Das TopLevel-Element des Objektmodells von Word ist der Objekttyp `Application`. Ein Objekt diesen Typs entspricht einer laufenden Instanz der Anwendung Word. Es besitzt unter vielen anderen ein Feld `ActiveDocument` für den Zugriff auf das gerade aktive Dokument und ein Feld `Selection`, um auf das gerade selektierte Element eines Dokuments zuzugreifen. Außerdem besitzt es ein Feld `Documents` mit einer Collection `Documents`, in der es die aktuell geöffneten Word-Dokumente verwaltet.

6.1.1 Die Collection Documents

Die Collection `Documents` bietet die üblichen Möglichkeiten einer Collection, darunter diese:

- Anzahl der geöffneten Dokumente abrufen mit `Documents.Count`,
- auf aktuell geöffnete Dokumente zugreifen per Index oder per Name,
- alle geöffneten Dokumente mit einer `ForEach`-Schleife durchlaufen.

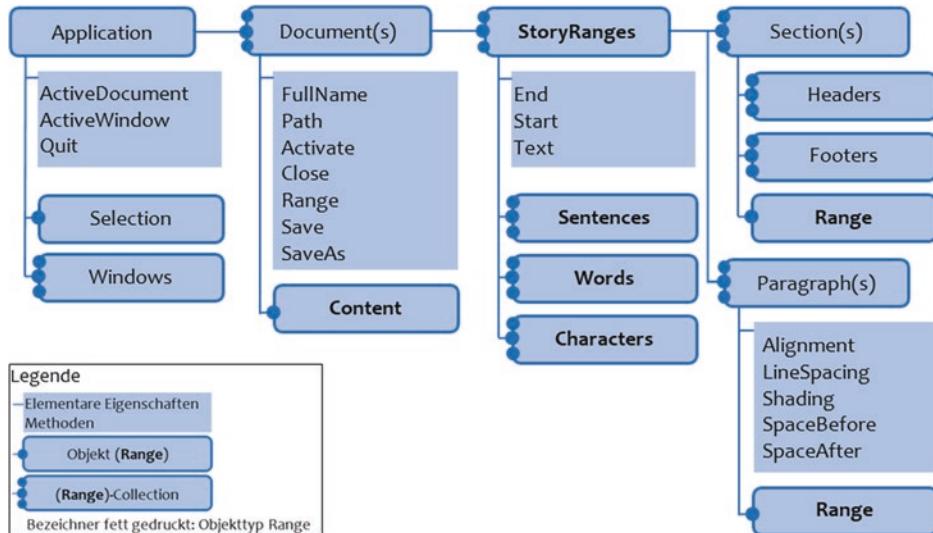


Abb. 6.1 Ausschnitt aus dem Word-Objektmodell

Die Methode `Open` der Collection `Documents` öffnet ein existierendes Dokument, die Methode `Add` erstellt dagegen ein neues Dokument. Man kann einem Word-Dokument nicht direkt einen Namen zuweisen, es erhält seinen Namen beim Speichern.

Das folgende kleine Demo-Skript läuft in Word als Wirtsanwendung. Es erzeugt ein Dokument, schreibt Text hinein, speichert es im Word- und im pdf-Format und schließt es. Vor dem Schließen zeigt es noch den Dateipfad des neuen Dokuments an.

```

Const DATEINAME = "test"

Sub DokumentErzeugen()
Dim dok As Document

Set dok = Documents.Add
dok.Content.InsertAfter "VBA-Objekte in Word"

dok.SaveAs2 DATEINAME & ".docx"
dok.ExportAsFixedFormat outputfilename:=DATEINAME & ".pdf", _
  exportformat:=wdExportFormatPDF
MsgBox dok.FullName
dok.Close
End Sub
  
```

6.1.2 Stories

Das Word-Objektmodell verwaltet den Inhalt eines Dokuments in Form von sogenannten Stories. Ein Word-Dokument hat den Objekttyp `Document`. Dieses besitzt ein Feld `StoryRanges`, das eine Collection mit den Stories des Dokuments speichert. Eine Story repräsentiert einen Inhalt des Dokuments. Sie enthält Text in Form einer Sequenz von sichtbaren, nicht-sichtbaren und Formatierungszeichen wie Buchstaben, Ziffern, Satzzeichen, Leerzeichen, Tabstopps, Absatzmarken usw.

Ein Dokument kann verschiedene Arten von Stories enthalten. Der Hauptinhalt des Dokuments ist eine dieser Stories. Wenn das Dokument Fußnoten enthält, sind diese in einer weiteren Story zusammengefasst. Die Kopfzeilen und Fußzeilen jedes Dokumentabschnitts bilden ebenfalls je eine Story. Insgesamt kennt Word 18 verschiedene `StoryTypes`. Mit `StoryRanges (wdMainTextStory)` erhält man den Hauptinhalt des Dokuments. Alternativ kann man auf das Feld `Content` eines `Document`-Objekts zugreifen, das ebenfalls den Hauptinhalt des Dokuments liefert. Das Word-Objektmodell enthält keinen eigenen Objekttyp für Stories, sondern verwendet dafür den Objekttyp `Range`, der noch an vielen weiteren Stellen des Objektsmodells vorkommt.

6.1.3 Der Objekttyp Range

Der wichtigste Objekttyp, um mit dem Text eines Word-Dokuments zu arbeiten, ist `Range`. Ein `Range`-Objekt beruht auf einer Story und repräsentiert einen zusammenhängenden Bereich in dieser Story. Der Bereich kann ein oder mehrere Zeichen umfassen oder auch leer sein kann.

Ein `Range`-Objekt auf dem Hauptinhalt eines Dokuments lässt sich mit der Methode `Range` des Objekttyps `Document` definieren. Man gibt dabei eine Anfangs- und eine Endzeichenposition an und erhält ein `Range`-Objekt, das einen Bereich im Hauptinhalt des Dokuments darstellt. Wenn Anfangs- und eine Endzeichenposition gleich sind, enthält das `Range`-Objekt keine Zeichen. Es entspricht dann einer Cursorposition zwischen zwei Zeichen.

Tab. 6.1 zeigt Beispiele für `Range`-Objekte und Abb. 6.2 veranschaulicht sie. Die Variable `dok` verweist dabei auf ein `Document`-Objekt, das mit dem Text „VBA-Objekte in Word“ beginnt. In Abb. 6.2 sind die möglichen Cursorpositionen zwischen den Zeichen dieses Texts markiert. Tab. 6.1 zeigt Aufrufe der Methode `Range` dieses Dokuments. Jeder der Aufrufe liefert ein `Range`-Objekt mit dem Text, der in der Tabelle aufgeführt ist.

Ein `Range`-Objekt ist definiert durch eine Anfangs- und eine Endzeichenposition, die es in seinen Feldern `Start` und `End` speichert, und durch die Story, die ihm zugrunde liegt. Sein Feld `Text` speichert den Text des Bereichs ohne Formatierungen als String. Neben `Start`, `End` und `Text` besitzt der Objekttyp `Range` noch viele weitere Felder.

Tab. 6.1 Mit der Methode Document.Range erzeugte Range-Objekte

Befehle	Text
dok.Range(0, 19)	VBA-Objekte in Word
dok.Range(0, 3)	VBA
dok.Range(1, 2)	B
dok.Range(0, 0)	<i>Kein Text, Cursorposition am Textanfang</i>
dok.Range(2, 2)	<i>Kein Text, Cursorposition zwischen B und A</i>

**Abb. 6.2** Cursor-Positionen bei der Definition von Range-Objekten

Einige Felder speichern Formatierungen und Einstellungen. In weiteren Felder stellt das Range-Objekt Informationen über die Struktur seines Inhalts bereit, die auch die Navigation im Dokument vereinfachen:

- Die Felder Characters, Words und Sentences enthalten Collections, in denen die einzelnen Zeichen, Wörter beziehungsweise Sätze innerhalb eines Range-Objekts aufgelistet sind. Die Elemente der Collections Characters, Words und Sentences sind selbst wieder Range-Objekte. Es gibt keine speziellen Objekttypen für Zeichen, Sätze oder Wörter.
- Das Feld Paragraphs speichert eine Collection Paragraphs mit Objekten des Typs Paragraph. Ein Paragraph-Objekt speichert einen Absatz des Word-Dokuments, also einen Bereich zwischen zwei Absatzmarken. Es enthält ein Range-Objekt und zusätzlich Felder für Absatzformate wie etwa Zeilenabstand, zentrierte, rechts- oder linksbündige Ausrichtung usw.
- Das Feld Sections speichert eine Collection Sections mit Objekten des Typs Section. Ein Section-Objekt repräsentiert einen Dokumentabschnitt, also einen Bereich zwischen zwei Abschnittsumbrüchen im Word-Dokument. Es enthält ebenfalls ein Range-Objekt und speichert zusätzlich noch Layout-Informationen wie Seitenränder, ein- oder mehrspaltigen Seitenaufbau etc.

In Abb. 6.1 sind alle Felder, die Range-Objekte oder Collections von Range-Objekten aufnehmen, mit Fettdruck beschriftet. Auch die Elemente der Collections Headers und Footers besitzen Felder für Range-Objekte, auch wenn die Abb. 6.1 diese nicht zeigt. Alle Range-Objekte besitzen ebenfalls die Felder, die in Abb. 6.1 ausgehend von der Collection StoryRanges eingezeichnet sind. So erscheint ein Word-Document im Word-Objektmodell als eine tief verschachtelte Struktur von Range-Objekten.

Methoden, mit denen der Inhalt eines Dokuments bearbeitet werden kann, sind im Objekttyp Range angesiedelt. Der folgende Abschnitt zeigt mit Code-Beispielen,

wie man sie einsetzt. Für das Verständnis ist es sehr hilfreich, sich klarzumachen, dass Range-Objekte (außer Stories) keine eigenständigen Text-Inhalte darstellen. Sie markieren lediglich Bereiche einer zugrunde liegenden Story.

6.1.4 Text ersetzen, einfügen und löschen

Einfüge-, Ersetzungs- und Löschoperationen werden mit Range-Objekten ausgeführt, wirken aber auf eine Story des Dokuments. Ein Range besitzt ein Feld `Text`, das den einfachen, unformatierten Text als String beinhaltet. Um diesen Text durch einen anderen Text zu ersetzen, weist man dem Feld `Text` einen neuen String zu, wie hier: `Range.Text = "neuer Text"`.

Der Objekttyp Range bietet auch mehrere `Insert`-Methoden, darunter diese:

- `Range.InsertBefore ("neuer Text")` fügt Text vor dem Start des Range ein.
- `Range.InsertAfter ("neuer Text")` fügt Text am End-Index des Range ein.

Wichtig ist dabei, dass diese Befehle den Range verändern. Der Range wird dabei so vergrößert, dass er auch den neuen Text umfasst. Sein `Start-Index` bleibt gleich, der `End-Index` erhöht sich um die Anzahl der Zeichen des neuen Texts.

Die Methode `Range.Delete` eines Range-Objekts löscht Text. Man kann sie auf zwei Arten verwenden:

- Wenn der Range Zeichen enthält, löscht `Delete` diese Zeichen. Parameter haben keine Wirkung.
- Für einen Range ohne Zeichen, also eine Cursorposition, akzeptiert `Delete` zwei Parameter: `Unit` und `Count`. `Unit` gibt die Einheit an, zum Beispiel Zeichen oder Wort. Die Einheit wird mit Konstanten spezifiziert, zum Beispiel mit `wdCharacter` beziehungsweise `wdWord`. Der Parameter `Count` gibt die Anzahl der zu löschen Einheiten an. Mit einem positiven Wert für `Count` löscht `Delete` die angegebene Anzahl Einheiten nach dem Range. Mit einem negativen Wert löscht `Delete` die angegebene Anzahl Einheiten vor dem Range. Beide Parameter sind optional. Voreinstellungen sind `wdCharacter` beziehungsweise 1.

Auch die Methode `Range.Collapse` wird oft gebraucht. Sie reduziert einen Range auf eine Cursorposition:

- `Range.Collapse (wdCollapseStart)` reduziert `Range(Start, End)` auf `Range(Start, Start)`
- `Range.Collapse (wdCollapseEnd)` reduziert `Range(Start, End)` auf `Range(End, End)`

Tab. 6.2 Beispiel für die Änderung eines Range beim Einfügen und Löschen von Text

Befehl	Start	End	Text
Set bereich = dok.Range(0, 0)	0	0	
.InsertAfter "Objekte"	0	7	Objekte
.InsertBefore "VBA-"	0	11	VBA-Objekte
.InsertAfter " in Word"	0	19	VBA-Objekte in Word
.Words(4).Text = "mit "	0	20	VBA-Objekte mit Word
.Collapse wdCollapseEnd	20	20	
.Delete wdWord, -2	12	12	
.SetRange 4, 11	4	11	Objekte
.Delete	4	4	

Das folgende Code-Beispiel setzt diese Methoden ein. Es läuft in der Wortsanwendung Word. Es erzeugt ein neues Dokument, das wie jedes neu erzeugte Dokument automatisch bereits einen Absatz enthält. Dessen einziges Zeichen ist eine (möglicherweise nicht sichtbare) Absatzmarke ¶. Der erste Befehl des Skripts definiert einen Range für die Cursorposition am Anfang des Dokuments. Dann fügt das Skript mit mehreren Befehlen Text ein und löscht ihn wieder. Diese Befehle ändern den Range. Tab. 6.2 zeigt die Werte der Felder Start, End und Text des Range nach Ausführung jedes Befehls.

```
Sub RangeDemo()
Dim dok As New Document
Dim bereich As Range
Set bereich = dok.Range(0, 0)
Debug.Print dok.Content.Start      ' 0
Debug.Print dok.Content.End        ' 1
With bereich
    .InsertAfter "Objekte"
    .InsertBefore "VBA-"
    .InsertAfter " in Word"
    .Words(4).Text = "mit "
    .Collapse wdCollapseEnd
    .Delete wdWord, -2
    .SetRange 4, 11
    .Italic = True
    .Delete
End With
dok.Close
End Sub
```

Wenn man das Code-Beispiel in der Entwicklungsumgebung in Einzelschritten ausführt und die Entwicklungsumgebung und das Word-Dokument nebeneinander platziert, kann man die Wirkung jedes einzelnen Befehls im Word-Dokument beobachten. Für die Einzelschritt-Ausführung platziert man den Cursor in der Entwicklungsumgebung im Code-Modul innerhalb der Sub RangeDemo und betätigt für jeden Befehl die Taste **F8**.

Das Skript zeigt weiterhin die Anwendung der Range-Methode SetRange, welche es erlaubt, Start und End eines Range neu festzulegen. Direkte Zuweisungen neuer Werte an die Eigenschaften Start und End eines Range-Objekts funktionieren nicht. Der folgende Abschnitt präsentiert eine Demo-Anwendung, die mithilfe dieser Befehle Dokumente erstellt.

6.2 Demo-Anwendung „Word aus Excel“

Nicht selten gibt es die Anforderung, aus einer Auswahl vorhandener Textteile individuelle Dokumente zusammenzustellen wie zum Beispiel Datenblätter oder Handbücher für konfigurierte Produkte, Angebotslisten oder auch Speisepläne. Dies lässt sich auf vielfältige Art mit Word umsetzen, zum Beispiel mit Textbausteinen und Dokumentvorlagen. Die hier vorgestellte Demo-Anwendung folgt einem anderen Ansatz. Sie verwaltet die Textteile in Excel und konfiguriert daraus mit VBA ein individuelles Word-Dokument. Der Vorteil von Excel als Speicher ist, dass die Textteile übersichtlich und leicht zugänglich bereitliegen, Formatierungen sind allerdings nicht möglich. Weiterhin ist nachteilig, dass längere Texte in Excel sehr unbequem zu editieren sind. Deshalb bietet die Demo-Anwendung auch ein Skript, um Textteile aus einem Word-Dokument in ein Excel-Arbeitsblatt zu übertragen, wenn Aktualisierungen nötig sind. Formatierungen würden dabei jedoch verloren gehen.

Die Textteile der Demo-Anwendung sind Produktbeschreibungen, genau genommen handelt es sich um überarbeitete Versionen von Daten aus [1]. Jedes Produkt hat einen Namen und einen beschreibenden Text und ist einer Kategorie zugeteilt. Abb. 6.3 zeigt die Texte im Word-Format. Jede Kategorie ist als Überschrift der Ebene 1 formatiert und jeder Produktnamen als Überschrift der Ebene 2. Auf den Produktnamen folgt die Produktbeschreibung, die aus mehreren Absätzen bestehen kann.

Abb. 6.4 zeigt Beispieldateien für die Demo-Anwendung im Excel-Format. Kategorien stehen dunkel unterlegt in der Spalte 2 einer sonst leeren Zeile. Jedes Produkt steht in einer eigenen Zeile, der Produktname in Spalte 2 und die Produktbeschreibung in Spalte 3. Die Spalte 1 bietet Platz, um die Produkte zu markieren, die in das Word-Dokument übernommen werden sollen. Dies ist im Code jedoch nicht umgesetzt.

Die Demo-Anwendung „Word aus Excel“ besteht aus zwei Teilen:

- „Excel2Word“ erstellt ein Word-Dokument aus Textteilen, die in Excel gespeichert sind.
- „Word2Excel“ überträgt Textteile aus Word in ein Excel-Arbeitsblatt.

The screenshot shows a Microsoft Word document window titled "modelle.docx - Word". The ribbon menu includes "Datei", "Start", "Einfüg", "Entwu", "Layout", "Verwe", "Sendu", "Überp", "Ansicht", "Entwick", "Zoterc", "Sie wünsch", "Anmelden", and "Freigeben". A search bar is present at the top right. The main content area displays a table of car models under sections "Oldtimer" and "Klassiker".

	Oldtimer	
1	1936 Mercedes-Benz 500K Spezial Roadster	Dieser Nachbau im Maßstab 1:18 ist aus schwerem Metalldruckguss gefertigt und verfügt über alle Merkmale des Originals: funktionierende Türen und Rumpelsitz, Einzelradaufhängung, detaillierter Innenraum, funktionierende Lenkung und eine aufklappbare Motorhaube, die einen so originalgetreuen Motor enthüllt, dass sogar die Verkabelung enthalten ist.
2	All dies wird durch eine Einbrennlackierung in Weiß abgerundet.	
3	1917 Grand Touring Limousine	Sehr detailliertes Modell des Plymouth Cuda von 1970 im Maßstab 1:12. Der Cuda gilt als einer der schnellsten originalen Muscle Cars der 1970er Jahre. Dieses Modell ist eine Replikation eines der originalen, im Jahr 1970 gebauten 652 Autos. Rote Farbe.
4		
5	Klassiker	
6	1970 Plymouth Hemi Cuda	Merkmale dieses Modells sind drehbare Vorderräder, echte Lenkfunktion, ein fein ausgearbeiteter Innenraum und ein detailliertes Fahrgestell. Türen, Kofferraum und Motorhaube lassen sich öffnen.
7	1952 Alpine Renault 1300	Drehbare Vorderräder, Lenkfunktion, detaillierter Innenraum, detaillierter Motor, zu öffnende Motorhaube, zu öffnender Kofferraum, zu öffnende Türen und detailliertes Fahrgestell.

Page navigation: Seite 1 von 2 | 318 Wörter | Deutsch (Deutschland)

Abb. 6.3 Anwendungsbeispiel „Dokumente transformieren zwischen Excel und Word“ – Word-Version

The screenshot shows a Microsoft Excel spreadsheet titled "modelle.xlsxm - Excel". The ribbon menu includes "Datei", "Start", "Einfüge", "Seitenl", "Formel", "Daten", "Überp", "Ansicht", "Entwick", "ACROB", "Sie wünsch", "Anmelden", and "Freigeben". The table structure is identical to the one in the Word document.

	Oldtimer	
5	1936 Mercedes-Benz 500K Spezial Roadster	Metalldruckguss gefertigt und verfügt über alle Merkmale des Originals: funktionierende Türen und Rumpelsitz, Einzelradaufhängung, detaillierter Innenraum, funktionierende Lenkung und eine aufklappbare Motorhaube, die einen so originalgetreuen Motor enthüllt, dass sogar die Verkabelung enthalten ist.
6	1917 Grand Touring Limousine	im Maßstab 1:12. Der Cuda gilt allgemein als einer der schnellsten originalen Muscle Cars der 1970er Jahre. Dieses Modell ist eine Replikation eines der originalen, im Jahr 1970 gebauten 652 Autos. Rote Farbe.
7	Klassiker	
8	1970 Plymouth Hemi Cuda	Merkmale dieses Modells sind drehbare Vorderräder, echte Lenkfunktion, ein fein ausgearbeiteter Innenraum und ein detailliertes Fahrgestell. Türen, Kofferraum und Motorhaube lassen sich öffnen.
9	1952 Alpine Renault 1300	Drehbare Vorderräder, Lenkfunktion, detaillierter Innenraum, detaillierter Motor, zu öffnende Motorhaube, zu öffnender Kofferraum, zu öffnende Türen und detailliertes Fahrgestell.

Sheet navigation: Modelle | Bereit | + | 85 %

Abb. 6.4 Anwendungsbeispiel „Dokumente transformieren zwischen Excel und Word“ – Excel-Version

Die folgenden Abschnitte zeigen den Code.

6.2.1 Teil „Excel2Word“ der Demo-Anwendung

Dieser Teil der Demo-Anwendung läuft in Excel als Wirtsanwendung und benötigt einen Verweis auf die Word-Objektbibliothek. Er besteht aus einer einzigen Prozedur.

Die Prozedur Excel2Word erwartet die Textteile, aus denen sie das Word-Dokument erzeugt, auf dem Arbeitsblatt „Modelle“ in der Excel-Arbeitsmappe, die auch das Skript enthält. Dateinamen und Pfad der erzeugten Dokumente sind hier fest eingespielt, damit der Code kompakt bleibt. Eine real zu nutzende Anwendung könnte sie aus einem Konfigurations-Arbeitsblatt lesen.

Die Prozedur führt folgende Aktionen aus: Sie startet eine Word-Application, erzeugt ein neues Word-Dokument und fügt darin die Inhalte aus dem Excel-Arbeitsblatt ein. Dann bietet sie an, das erzeugte Dokument zu speichern, und schließt Word. Sie beinhaltet weiterhin eine Fehlerbehandlung.

```
Sub Excel2Word()
    Dim blatt As Worksheet
    Dim zeile As Long
    Dim wApp As New Word.Application
    Dim dok As Word.Document

    Set blatt = Worksheets("Modelle")

    wApp.Visible = True
    Set dok = wApp.Documents.Add()

    On Error GoTo Fehler
    zeile = 1
    With dok.Paragraphs
        Do While blatt.Cells(zeile, 2) <> ""
            .Last.Range.InsertBefore blatt.Cells(zeile, 2).Value
            If Cells(zeile, 3).Value = "" Then
                .Last.Format.Style = "Überschrift 1"
                .Add
            Else
                .Last.Format.Style = "Überschrift 2"
                .Add
            End If
            .Last.Format.Style = "Standard"
            .Last.Range.InsertBefore blatt.Cells(zeile, 3).Value
        Loop
    End With
Fehler:
    wApp.Quit
End Sub
```

```
.Add
End If
zeile = zeile + 1
Loop
End With

With ThisWorkbook
If MsgBox("Speichern?", vbYesNo, "Excel -> Word") Then
    dok.SaveAs Left(.FullName, Len(.FullName) - 5) & ".docx"
End If
End With
GoTo Aufraeumen

Fehler:
Debug.Print Err.Source, Err.Description

Aufraeumen:
dok.Close SaveChanges:=False
wApp.Quit
Set wApp = Nothing
End Sub
```

Der Code zum Einfügen der Inhalte ist übersichtlich: Das Skript arbeitet mit dem Word-Objekttyp `Paragraph`. Ein frisch erzeugtes Word-Dokument besteht aus einem leeren Absatz. Im Feld `Paragraphs.Last` kann man auf einfache Weise auf diesen Absatz zugreifen. Der Code durchläuft in einer Schleife die Zeilen des Excel-Arbeitsblatts und fügt für jede nicht leere Zelle vor dem letzten Absatz einen neuen Absatz in das Dokument ein. Dann fügt er den Zelleninhalt aus Excel ein und formatiert den Absatz. Der ursprünglich vorhandene, leere letzte Absatz bleibt dabei am Ende des Dokuments erhalten und könnte, wenn nötig, entfernt werden.

Die Möglichkeit, Markierungen, zum Beispiel „x“, aus der Spalte 1 auszulesen und nur markierte Produktbeschreibungen nach Word zu übertragen, lässt sich in der Schleife auf einfache Weise ergänzen.

6.2.2 Teil „Word2Excel“ der Demo-Anwendung

Dieser Teil der Demo-Anwendung ist in Word als Wortsanwendung implementiert und benötigt einen Verweis auf die Excel-Objektbibliothek. Er besteht aus einer Hauptprozedur und einer Hilfsprozedur. Die Hauptprozedur `Word2Excel` speichert die Textinhalte aus dem Word-Dokument in ein Excel-Arbeitsblatt. Sie ist ähnlich aufgebaut wie ihr Gegenstück `Excel2Word`.

Die Prozedur Word2Excel erzeugt eine neue Excel-Arbeitsmappe, die automatisch bereits ein Arbeitsblatt enthält, und stellt darauf einige Formatierungen ein. Dann überträgt sie in einer Schleife die Textteile aus dem Word-Dokument in die Zellen des Excel-Arbeitsblatts. Schließlich bietet sie an, die erzeugte Arbeitsmappe zu speichern, und beendet Excel.

Die Schleife läuft durch die Absätze des Word-Dokuments und erhöht dabei einen Index für die Zeilen des Excel-Arbeitsblatts. Sie erkennt die Art des Absatzes an seiner Formatierung: Ein als Überschrift 1 formatierter Absatz ist eine Kategoriebezeichnung und gehört in die Spalte 2. Ein als Überschrift 2 formatierter Absatz ist eine Produktbezeichnung und gehört ebenfalls in die Spalte 2. Solange darauffolgende Absätze weder als Überschrift 1 noch als Überschrift 2 formatiert sind, gehören sie zur Produktbeschreibung. Die Prozedur sammelt diese Absätze und fügt sie aneinander. Erst wenn sie wieder auf eine Überschrift oder auf das Dokumentende trifft, schreibt sie die gesammelten Absätze der Produktbeschreibung in die Spalte 3 der aktuellen Zeile.

```
Sub Word2Excel ()  
Dim exApp As New Excel.Application  
Dim dok As Word.Document  
Dim blatt As Worksheet  
Dim mappe As Workbook  
Dim sammel As String  
Dim zeile As Long  
Dim index As Long  
Dim absatz As Word.Paragraph  
  
exApp.Visible = True  
Set mappe = exApp.Workbooks.Add  
Set blatt = mappe.Worksheets(1)  
Set dok = ThisDocument  
  
blatt.Columns(2).ColumnWidth = 39  
blatt.Columns(3).ColumnWidth = 50  
blatt.Columns(3).WrapText = True  
zeile = 1  
sammel = ""  
  
On Error GoTo Fehler  
For index = 1 To dok.Paragraphs.Count - 1  
    Set absatz = dok.Paragraphs(index)  
    If absatz.Format.Style = "Überschrift 1" Then  
        blatt.Cells(zeile, 2).Value = AbsEntfernen(absatz.Range.text)  
        blatt.Cells(zeile, 2).Interior.Color = wdColorAqua  
        blatt.Cells(zeile, 3).Interior.Color = wdColorAqua
```

```
zeile = zeile + 1
ElseIf absatz.Format.Style = "Überschrift 2" Then
    blatt.Cells(zeile, 2).Value = AbsEntfernen(absatz.Range.text)
ElseIf absatz.Next.Format.Style = "Überschrift 1" _
    Or absatz.Next.Format.Style = "Überschrift 2" _
    Or index = dok.Paragraphs.Count - 1 _
    Then
        sammel = sammel & absatz.Range.text
        blatt.Cells(zeile, 3).Value = AbsEntfernen(sammel)
        sammel = ""
        zeile = zeile + 1
    Else
        sammel = sammel & absatz.Range.text
    End If
Next
exApp.DisplayAlerts = False

If (vbYes = MsgBox("Speichern?", vbYesNo)) Then
    mappe.SaveAs Left(dok.FullName, Len(dok.FullName) - 5) & ".xlsx"
End If

GoTo Aufraeumen

Fehler:
Debug.Print Err.Source, Err.Description

Aufraeumen:
mappe.Close SaveChanges:=False
exApp.Quit
Set exApp = Nothing
End Sub

Function AbsEntfernen(text As String) As String
If text = "" Then
    AbsEntfernen = text
Else
    If Right(text, 1) = vbLf Or Right(text, 1) = vbCr _
    Or Right(text, 1) = vbCrLf Then
        AbsEntfernen = AbsEntfernen(Left(text, Len(text) - 1))
    Else
        AbsEntfernen = text
    End If
End If
End Function
```

Etwas Aufwand erfordert hier, die Absatzmarken zu behandeln. Das Skript soll Absatzmarken, die am Ende einer Produktbeschreibung stehen, nicht nach Excel übernehmen. Absatzmarken innerhalb der Produktbeschreibung soll es dagegen beibehalten. Dazu dient die rekursive Hilfsfunktion `AbsEntfernen`. Sie löscht alle Zeichen, die einen Zeilenumbruch bewirken, vom Ende eines Strings.

6.3 Textmarken und Tabellen

Eine Textmarke (Bookmark) definiert einen Range und gibt ihm einen Namen, sodass ein VBA-Skript auf den Range zugreifen kann. Textmarken bieten einen guten Ansatz, um wechselnde Textteile in ein Dokument einzufügen. Das vorbereitete, fertig formulierte Dokument enthält dabei Textmarken als Platzhalter. Ein VBA-Skript kann dann die Textmarken mit wechselnden Inhalten ersetzen.

Bei diesem Ansatz produziert nicht das VBA-Skript Formatierungen, Layout und gleichbleibende Textteile, sondern sie sind manuell in Word erstellt und bei Bedarf leicht zu modifizieren. Dies bringt Flexibilität und hält den VBA-Code schlank.

Word verwaltet die Textmarken eines Dokuments in einer Collection `Bookmarks`. Eine neue Textmarke lässt sich mit der Methode `Add` der Collection `Bookmarks` definieren. Als Argumente benötigt die Methode einen Namen sowie den Bereich, den die Textmarke umfassen soll.

Der Objekttyp `Bookmark` besitzt die zwei Felder `Name` und `Range`. Der Text der Textmarke kann mit den Methoden ihres `Range`-Objekts bearbeitet werden, zum Beispiel:

- `Bookmark.Range.InsertBefore("Neuer Text")`
- `Bookmark.Range.InsertAfter("Neuer Text")`
- `Bookmark.Range.Text = "Neuer Text"`

Die `Insert`-Methoden ändern den Range der Textmarke so wie in Abschn. 6.1.4 beschrieben. Der Befehl `Bookmark.Range.Text = "Neuer Text"` ersetzt nicht nur den Text in der zugrunde liegenden Story, er löscht außerdem die Textmarke komplett aus dem Dokument. Wenn sie weiter benötigt wird, muss sie erneut definiert werden. Die Prozedur `TMAktualisieren` des folgenden Code-Beispiels demonstriert, wie dies einfach zu erreichen ist [2]: Sie speichert den Range der `Bookmark` in einer Variablen und aktualisiert seinen Text. Dann definiert sie eine neue `Bookmark` mit dem jetzt aktualisierten Range und dem alten Namen.

```
Sub TMAktualisieren(dok As Document, tmName As String, neu As String)
    Dim TMMerk As Range
    Set TMMerk = dok.Bookmarks(tmName).Range
    TMMerk.Text = neu
```

```
dok.Bookmarks.Add tmName, TMerk
End Sub

Sub TextmarkeTest()
Documents.Add().Activate
With ActiveDocument
    .Range.InsertAfter ("Einfach einige Wörter schreiben")
    .Bookmarks.Add "test", .Words(2)
    .Bookmarks("test").Range.Bold = True
    TMAktualisieren ActiveDocument, "test", "neue "
    .Bookmarks("test").Range.Italic = True
End With
End Sub
```

Die Prozedur `TextmarkeTest` wendet `TMAktualisieren` an. Sie erzeugt ein Dokument mit etwas Text und eine Textmarke, die das zweite Wort des Dokuments abdeckt. Dann führt sie die Prozedur `TMAktualisieren` aus, um den Text der Textmarke zu ersetzen. Eine Formatierung zeigt anschließend, dass die Textmarke existiert und welchen Bereich sie einnimmt.

Die Demo-Anwendung im folgenden Abschnitt verwendet neben Textmarken auch eine Word-Tabelle, um individuelle Inhalte in einem Word-Dokument zu positionieren. Der grundlegende Umgang mit Word-Tabellen ist einfach:

- Ein Word-Dokument verwaltet Tabellen in einer Collection `Tables`.
- Die Zellen einer einfachen Tabelle lassen sich mit Indizes adressieren, ähnlich zu den Zellen eines Excel-Arbeitsblatts.
- Eine Zelle speichert ihren Inhalt als `Range`-Objekt.

Beispiele dazu zeigt die Demo-Anwendung.

6.4 Demo-Anwendung „Produktblätter“

Die Demo-Anwendung „Produktblätter“ erstellt individuelle Produktblätter für verschiedene Produkte mithilfe von Textmarken und einer Tabelle. Ein Word-Dokument „datenblatt.docx“ bildet die Vorlage für ein Produktdatenblatt wie in Abb. 6.5. Es enthält Formatierungen und Textteile sowie zwei Textmarken als Platzhalter für den Namen und die Beschreibung eines Produkts. Für die Demo sind die Texte der Textmarken kursiv und in einer besonderen Schriftart formatiert. Eingefügte Produktdaten werden diese Formatierung übernehmen.

Weitere Produktdaten präsentiert das Datenblatt in einer Tabelle. Da das VBA-Skript eine Tabellenzelle über ihren Zeilen- und Spaltenindex adressieren kann, sind keine Textmarken erforderlich, um Daten in die Tabelle einzufügen. Das Erstellungsdatum ist mit

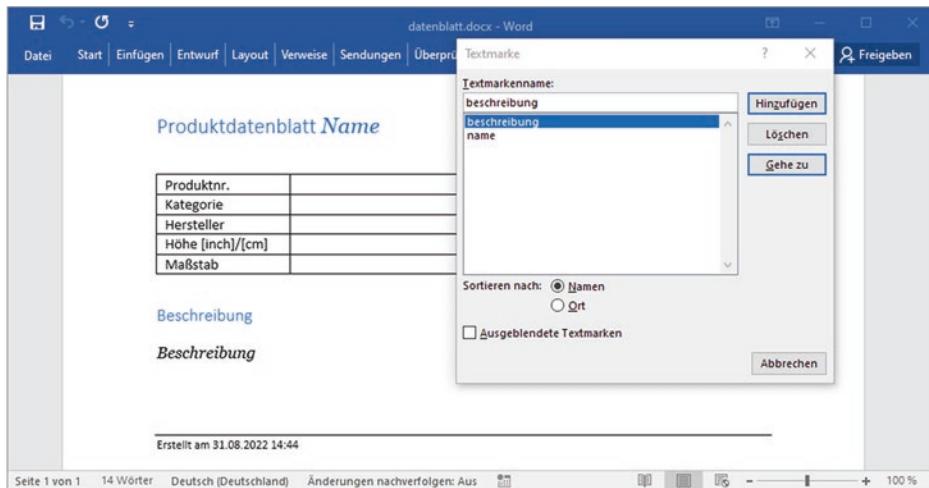


Abb. 6.5 Datenblatt-Vorlage für die Demo-Anwendung „Produktblätter“

dem Datumsfeld `SaveDate` aus dem Schnellbausteine-Katalog von Word realisiert und aktualisiert sich automatisch ohne VBA.

Die Demo-Anwendung erzeugt ein Produktdatenblatt, indem sie die Vorlage öffnet, mit Werten befüllt und unter einem neuen Dateinamen abspeichert. Die Vorlage bleibt dabei unverändert. Die Demo-Anwendung definiert und verwendet dabei einen eigenen Datentyp `Produkt`.

```

Private Type Produkt
    nummer As String
    name As String
    kategorie As String
    hersteller As String
    hoehe As Double
    masstab As String
    beschreibung As String
End Type

Sub DatenblattDemo()
    Dim dok As Document
    Dim p As Produkt
    Dim nr As Integer

    nr = InputBox("Welches Produkt? Bitte 1 oder 2 eingeben.")
    p = Produktdaten(nr)

```

```
Set dok = Documents.Open(ThisDocument.Path & "\datenblatt.docx")
dok.Bookmarks("name").Range.text = p.name
dok.Bookmarks("beschreibung").Range.text = p.beschreibung
With dok.Tables(1)
    .Cell(1, 2).Range.text = p.nummer
    .Cell(2, 2).Range.text = p.kategorie
    .Cell(3, 2).Range.text = p.hersteller
    .Cell(4, 2).Range.text = p.hoehe & " / " & p.hoehe * 2.54
    .Cell(5, 2).Range.text = p.massstab
End With

If vbYes = MsgBox("Speichern?", vbYesNo) Then
    dok.SaveAs2 ThisDocument.Path & "\prodblatt" & nr & ".docx"
End If
dok.Close SaveChanges:=False
End Sub

Function Produktdaten(nr As Integer) As Produkt
Dim p As Produkt
If nr = 1 Then
    p.nummer = "S10_1678"
    p.name = "1969 Harley Davidson Ultimate Chopper"
    p.kategorie = "Motorräder"
    p.hersteller = "Min Lin Diecast"
    p.hoehe = 1.77
    p.massstab = "1:10"
    p.beschreibung = "Dieser Nachbau hat einen funktionierenden " _
    & "Ständer, Vorderradaufhängung, Schalthebel, Fußbremshebel, " _
    & "Antriebskette, Räder und Lenkung."
Else
    p.nummer = "S10_1678"
    p.name = "1996 Moto Guzzi 1100i"
    p.kategorie = "Motorräder"
    p.hersteller = "Highway 66 Mini Classics"
    p.hoehe = 0.68
    p.massstab = "1:10"
    p.beschreibung = "Offizielle Moto Guzzi Logos und Insignien " _
    & "sowie Satteltaschen an der Seite. Ein detailreicher Motor, " _
    & "eine funktionstüchtige Lenkung und Federung, drehbare Räder " _
    & "und ein funktionierender Ständer begeistern. Das Modell " _
    & "hat zwei Ledersitze, zwei Auspuffrohre und eine Lenkertasche."
End If
Produktdaten = p
End Function
```

Die Daten der einzelnen Produkte können bei einer solchen Anwendung zum Beispiel aus einer Datenbank oder aus einer Excel-Arbeitsmappe stammen. Die Demo-Anwendung erzeugt Beispieldaten für zwei Produkte durch ein Skript, damit sie auf eine solche externe Datenquelle verzichten kann. Dies leistet die Funktion `Produktdaten`. Sie demonstriert dabei, wie man eine Variable eines selbstdefinierten Type mit Daten befüllt und als Resultat einer Funktion ausgibt.

6.5 Dokumente automatisiert bearbeiten mit Suchen und Ersetzen

VBA-Skripte können auf verschiedene Weise helfen, die Arbeit mit Dokumenten effizienter zu gestalten, zum Beispiel auch beim Bereinigen und Vereinheitlichen umfangreicher oder vieler Dokumente. Die folgenden Code-Beispiele nutzen die Such-Funktion von Word, um Formatierungsarbeiten zu automatisieren. Im Word-Objektmodell sind die Methoden für Suchen und Ersetzen im Objekttyp `Range` angesiedelt, sodass sich Such- und Ersetzvorgänge auf bestimmte Bereiche eines Dokuments beschränken lassen.

6.5.1 Der Objekttyp `Find`

Die „Erweiterte Suche“ von Word bietet viele Einstellmöglichkeiten, die auch in VBA zur Verfügung stehen. Man kann sie in einem `Find`-Objekt konfigurieren, wie die folgenden Code-Beispiele zeigen. Der Suchtext wird im Feld `Text` des `Find`-Objekts hinterlegt.

Ein `Find`-Objekt gehört zu einem `Range`-Objekt. Wenn dieser `Range` mehrere Zeichen umfasst, sucht `Find` den Suchtext innerhalb des `Range`. Wenn `Start` und `End` des `Range`-Objekts identisch sind, sucht `Find` den Suchtext im Dokument ab der Position des `Range`.

Die `Execute`-Methode des `Find`-Objekts startet die Suche. Sie liefert als Ergebnis einen Booleschen Wert. Ein erfolgloser Suchvorgang endet mit dem Ergebnis `False`. Ein erfolgreicher Suchvorgang liefert das Resultat `True` und verändert den `Range`: er wird auf den gefundenen Suchtext reduziert. Dies ist ganz praktisch, denn jetzt kann ein Skript die gefundene Textstelle im `Range`-Objekt bearbeiten.

Wenn ein Skript alle Vorkommen eines Texts in einem Bereich finden soll, muss es die `Execute`-Methode des `Find`-Objekts so oft anstoßen, bis diese schließlich mit dem Ergebnis `False` endet. Dabei muss es nach jedem erfolgreichen Suchvorgang auch den Suchbereich wieder anpassen. Auch dies wird in den Code-Beispielen gezeigt.

6.5.2 Der Objekttyp Selection

Ein Objekt `Selection` (= Auswahl) repräsentiert einen Bereich eines Word-Dokuments, der aktuell ausgewählt ist. Das Code-Beispiel im folgenden Abschn. 6.5.3 zeigt, wie ein VBA-Skript mit dem `Selection`-Objekt auf die aktuelle Auswahl zugreifen kann, um sie zu durchsuchen und zu bearbeiten. Dies kann sehr nützlich sein, denn Benutzer können so den Bereich des Dokuments, den ein Skript bearbeiten soll, einfach mit der Maus markieren.

Word kann ein Dokument in mehreren Fenstern anzeigen. Das `Application`-Objekt verwaltet die Fenster in seiner Collection `Windows`. Sein Feld `ActiveWindow` verweist auf das gerade aktive Fenster. Ein Fenster hat den Objekttyp `Window`. Im Zusammenhang mit `Selection` ist dies von Bedeutung, weil jedes Word-Fenster ein eigenes `Selection`-Objekt besitzt. Ein VBA-Skript greift deshalb über ein `Window`-Objekt auf eine `Selection` zu, nicht etwa über ein `Document`.

Die Skripte, die der Makrorecorder von Word aufzeichnet, arbeiten typischerweise mit dem `Selection`-Objekt. Ein `Selection`-Objekt besitzt ein Feld `Range` und ein Feld `Text`, die den gerade markierten Bereich und seinen Text enthalten. Das Feld `Text` ist im Objekttyp `Selection` voreingestellt. Der Ausdruck `Selection`, der in den aufgezeichneten Skripten des Word-Makrorecorders häufig vorkommt, ist somit die Kurzform von `ActiveWindow.Selection.Text`.

Die Objekttypen `Selection` und `Range` sind sehr ähnlich. Da sich aber `Selection`-Objekte in bestimmten Fällen nicht intuitiv verhalten, ist es im Allgemeinen besser, mit `Range`-Objekten arbeiten [3]. Für den Anwendungsfall des folgenden Code-Beispiels ist die `Selection` aber gut geeignet.

6.5.3 Code-Beispiel: Wörter innerhalb der Auswahl formatieren

Das hier vorgestellte Code-Beispiel formatiert alle Vorkommen eines Suchtext (hier „VBA“) neu, wenn sie bestimmte Bedingungen erfüllen: die Groß-Kleinschreibung muss mit dem Suchtext übereinstimmen, es muss sich um ein ganzes Wort handeln (nicht um eine Zeichenfolge innerhalb eines Worts) und es darf kein Bindestrich „-“ folgen. Es soll nur den als `Selection` markierten Bereich bearbeiten.

Ein erfolgreicher Suchvorgang verändert den durchsuchten `Range`. Damit der ursprünglich markierte Bereich dabei nicht verloren geht, dupliziert das Code-Beispiel zuerst den `Range` der `Selection` in ein weiteres `Range`-Objekt `bereich`. Dann parametrisiert es das `Find`-Objekt des `Range`-Objekts `bereich` und beginnt zu suchen.

Wenn ein Vorkommen des Suchtext gefunden und formatiert ist, reduziert das Skript den `Range` `bereich` auf die Cursorposition nach der letzten Fundstelle und sucht von dort aus weiter. Es startet die `Execute`-Methode des `Find`-Objekts in einer Schleife immer wieder neu, solange sie das Ergebnis `True` liefert und der Start des Suchbereichs noch vor dem Ende des im `Selection`-Objekt markierten `Range` liegt.

```
Const SUCHTEXT As String = "VBA"

Sub Formatieren()
Dim bereich As Range
Set bereich = Selection.Range.Duplicate      ' Find ändert Range

With bereich.Find
    .Text = SUCHTEXT
    .MatchCase = True          ' Groß-Kleinschreibung wie im Suchtext
    .MatchWholeWord = True     ' nur ganze Wörter
    .MatchWildcards = False
    .MatchSoundsLike = False
    .MatchAllWordForms = False
Do While .Execute = True And bereich.Start <= Selection.Range.End
    If bereich.InRange(Selection.Range) And _
        ActiveDocument.Characters(bereich.End + 1).Text <> "-" Then
        bereich.Font.Color = wdColorAqua
        bereich.Font.Name = "Arial"
        bereich.Font.Bold = True
        bereich.Collapse wdCollapseEnd
    End If
Loop
End With
End Sub
```

Das Code-Beispiel modifiziert die Suche noch mit zwei Besonderheiten:

Die Bedingung `bereich.InRange(Selection.range)` bewirkt, dass das Skript nur Vorkommen des Suchtextes bearbeitet, die komplett im markierten Bereich liegen.

Das Skript betrachtet nicht nur den eigentlichen Suchtext, sondern auch das Zeichen, das auf eine Fundstelle folgt, denn wenn es sich dabei um einen Bindestrich handelt, soll die gefundene Textstelle nicht bearbeitet werden. Das Folgezeichen einer Fundstelle ermittelt die Prozedur, indem sie auf die Collection `Characters` des `ActiveDocument` zugreift. Dies macht das Skript von der Umgebungssituation abhängig, denn es funktioniert nur dann wie beabsichtigt, wenn das `ActiveDocument` das zu bearbeitende Dokument ist. In der hier skizzierten Anwendungssituation kann man davon ausgehen, dass diese Voraussetzung zutrifft: wenn Benutzer vor dem Start des Skripts den Suchbereich interaktiv selektieren, haben sie damit zugleich auch das richtige Dokument aktiviert. Für andere Anwendungsszenarien, zum Beispiel wenn mehrere Dokumente in einem Massen-Update zu bearbeiten sind, würde man dieses Skript und auch

das folgende Code-Beispiel besser ohne den Zugriff auf ActiveDocument programmieren.

6.5.4 Code-Beispiel: Wörter mit bestimmtem Absatzformat formatieren

Das Code-Beispiel dieses Abschnitts dient ebenfalls der Formatierung. Es bearbeitet alle Vorkommen eines Suchtext im gesamten aktiven Dokument, sofern sie ein bestimmtes Absatzformat aufweisen. Das Skript lässt sich zum Beispiel für das Syntax-Highlighting von Programmcode in einem Dokument ausbauen, kann beim Formatieren Überschriften übergehen etc.

```
Const ABSATZFORMAT = "Standard"
Const SUCHTEXT = "VBA"

Sub VBAFormatieren()
Dim bereich As Range

Set bereich = ActiveDocument.Range
With bereich.Find
    .Text = SUCHTEXT
    .MatchCase = True
    .MatchWholeWord = True
    .MatchWildcards = False
    .MatchSoundsLike = False
    .MatchAllWordForms = False
    Do While .Execute = True
        If bereich.ParagraphFormat.Style = ABSATZFORMAT Then
            bereich.Font.Color = wdColorAqua
            bereich.Font.Name = "Arial"
            bereich.Font.Bold = True
            bereich.Collapse wdCollapseEnd
        End If
    Loop
End With
End Sub
```

Dieses Skript hat nur dann einen sichtbaren Effekt, wenn das bearbeitete Dokument Absätze mit dem Absatzformat „Standard“ und dem Suchtext enthält. Hier lassen sich andere Absatzformate einsetzen oder auch Bedingungen formulieren, die nicht das Absatzformat betreffen.

6.6 Demo-Anwendung „Etiketten-Tool“

Digitalisierung erfordert oft, reale physische Objekte in digitale Prozesse einzubeziehen. Ein bewährter Ansatz dafür sind Barcodes. Die folgende Demo-Anwendung erstellt Barcode-Etiketten für Inventar. Die Barcodes generiert es mit einer Funktion, die Word bereits mitbringt. Der Barcode auf einem Inventaretikett soll eine fortlaufende Inventarnummer darstellen. Eine reale Inventarverwaltung würde der Inventarnummer einen Datensatz mit Beschreibung, Einsatzort, Ausgabedatum, Wartungsinformationen und ähnlichem zuordnen, der etwa in einem Datenbanksystem oder bei kleinen Datenmengen in einer Excel-Arbeitsmappe verwaltet wird. Auch eine Anbindung des Etiketten-Tools an ein ERP-System oder ähnliche Anwendungssoftware wäre zweckmäßig.

Um die fortlaufende Inventarnummer zu generieren, muss das Etiketten-Tool den jeweils aktuellen Stand der Inventarnummerierung speichern. Ein Word-Dokument bietet keine offensichtliche Möglichkeit, um Daten wie etwa einen Nummernstand zu hinterlegen. Das hier vorgestellte Etiketten-Tool löst dies, indem es die jeweils aktuelle Inventarnummer in den Dokumenteneigenschaften sichert. Dies bringt den Vorteil, dass Anwender die Inventarnummer dort jederzeit einsehen und auch editieren können, siehe Abb. 6.6. Neben der Barcode-Funktion demonstriert die Demo-Anwendung „Etiketten-Tool“ damit auch, wie VBA eine Dokumenteneigenschaft anlegen, abrufen und aktualisieren kann. Außerdem zeigt sie Befehle, um das Seitenformat eines Dokuments festzulegen.

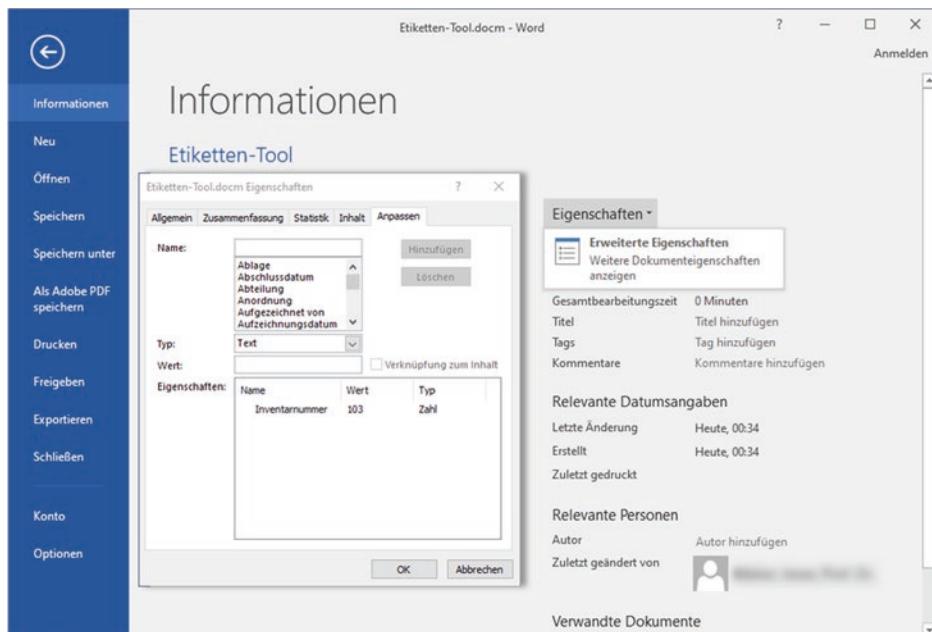


Abb. 6.6 Dokumenteneigenschaften

6.6.1 Der Objekttyp DocumentProperty

Ein Word-Dокумент verwaltet vordefinierte Dokumenteneigenschaften wie „Autor“, „Zeitpunkt der Erstellung“, „Zeitpunkt der letzten Änderung“ usw. in einer Collection in seinem Feld `BuiltinDocumentProperties`. Benutzerdefinierte Dokumenteneigenschaften sammelt es in einer Collection im Feld `CustomDocumentProperties`. Beide Collections haben den Objekttyp `DocumentProperties`. Die Dokumenteneigenschaften in den beiden Collections haben den Objekttyp `DocumentProperty` mit den Feldern `Name` und `Value`.

VBA-Code kann die Collections mit einer `ForEach`-Schleife durchlaufen oder mit Name oder Index auf eine einzelne Dokumenteneigenschaft zugreifen. Der folgende Code-Schnipsel demonstriert diese Zugriffsmöglichkeiten in der Wirtsanwendung Word. Laufzeitfehler treten auf, wenn der Code auf eine Dokumenteneigenschaft zugreift, die nicht existiert, und wenn er versucht, den Wert einer nicht gesetzten numerischen Dokumenteneigenschaft abzurufen. Der Code-Schnipsel produziert ausführliche Debug-Ausgaben, um alle Laufzeitfehler anzuzeigen.

```
Sub DemoDocProps()
Dim p As DocumentProperty
Dim props As DocumentProperties

Set props = ThisDocument.BuiltInDocumentProperties
' Set props = ThisWorkbook.BuiltInDocumentProperties   ' für Excel
On Error GoTo Info

For Each p In props
    Debug.Print p.Name & ":"
    Debug.Print " > " & p.Value
Next

Debug.Print "Author: " & props("Author").Value
Debug.Print "Erste Property: " & props(1).Name
Debug.Print "Property Gibtsnicht " & props("Gibtsnicht").Value
Debug.Print "Number of notes: " & props("Number of notes ").Value
Exit Sub

Info:
Debug.Print " ! "      ' einen Fehler anzeigen
Resume Next           ' weiterarbeiten
End Sub
```

Die Objekttypen `DocumentProperties` und `DocumentProperty` stammen nicht aus dem Objektmodell von Word, sie gehören zum Objektmodell von Office. Die-

ses ist in Office-Wirksanwendungen automatisch mit eingebunden, siehe zum Beispiel in Abb. 4.6. Das beschriebene Vorgehen funktioniert damit auch in anderen Office-Anwendungen wie Excel etc. Im Objektmodell von Excel etwa gehören die Felder `Built-inDocumentProperties` und `CustomDocumentProperties` dem Objekttyp `Workbook`.

6.6.2 Der Objekttyp Field

Word beinhaltet eingebaute Funktionen, die dynamisch Inhalte in Dokumenten generieren. Beispiele sind die Funktion DATE, die das aktuelle Datum liefert, die Funktion SEQ, die einen fortlaufenden Zähler realisiert, oder auch die Funktion TOC, die ein Inhaltsverzeichnis erzeugt. Um eine solche Funktion in einem Word-Dokument zu verwenden, fügt man ein Feld an der Stelle in das Dokument ein, an der das Ergebnis der Funktion erscheinen soll, und hinterlegt die Funktion in diesem Feld. Das Feld dient somit als Platzhalter für den dynamisch erzeugten Inhalt.

In der Bedienoberfläche von Word findet man Felder und Feldfunktionen im Menüband unter **Einfügen > Schnellbausteine > Feld....**. Im Word-Objektmodell hat ein Feld den Objekttyp `Field`. Ein Dokument verwaltet seine Felder in einer Collection `Fields`. Der Code der Demo-Anwendung erzeugt mit einem solchen Feld und einer Feldfunktion einen Barcode.

6.6.3 Erzeugen von Barcodes

Die Funktion `DISPLAYBARCODE` verwandelt Text in einen Barcode. Sie unterstützt mehrere Barcodetypen, darunter QR-Codes und CODE128. Zwei Beispiele für Formeln zur Generierung von Barcodes sind:

- `DISPLAYBARCODE "123.456" QR \q 3`
- `DISPLAYBARCODE "123.456" CODE128 \t`

Die Funktion erwartet zwei Argumente und akzeptiert zusätzliche optionale Schalter. Das erste Argument ist der Text, der als Barcode dargestellt werden soll. Dies kann beispielsweise eine URL sein oder auch eine Ziffernfolge. Im Fall des Etiketten-Tools bildet die Inventarnummer den Text. Wie lang der Text maximal sein darf und aus welchen Zeichen er bestehen kann, bestimmt der Barcode-Typ. Diesen legt das zweite Argument der Funktion fest.

Die Schalter sind optional und zum Teil nur für bestimmte Barcode-Typen anwendbar. Beispielsweise legt der Schalter `\q 3` die Fehlerkorrekturstufe für einen QR-Code auf den Wert 3 fest. `\t` besagt, dass die Funktion nicht nur die Barcode-Grafik, sondern

zusätzlich auch den Text anzeigen soll, in den Beispielen oben also die Zeichenfolge „123.456“. [4]

6.6.4 Code der Demo-Anwendung

Das folgende VBA-Skript realisiert ein einfaches Etiketten-Tool. Es ist für die Wirtschaftsanwendung Word konzipiert und besteht aus drei Konstantendeklarationen, einer Sub und einer Funktion. Man fügt alles in ein Code-Modul eines Word-Dokuments ein und konfiguriert in einer Konstanten einen Drucker. Weitere Daten oder Vorlagen sind für die Demo nicht erforderlich.

Das so erstellte Etiketten-Tool sollte bereits vor der ersten Ausführung als Word-Dokument mit Makros gespeichert sein. Dann stehen Dateiname und Speicherort bereits fest, wenn der Code das Etiketten-Tool nach der Ausführung mit aktualisiertem Nummernstand erneut sichert.

Die Sub `Etikett` erzeugt das Inventaretikett. Ihre erste wesentliche Aktion ist, mithilfe der Funktion `DokEigenschaft` den aktuellen Nummernstand aus den Dokumenteneigenschaften zu lesen. Der Name der Dokumenteneigenschaft und ein Initialwert sind in Konstanten hinterlegt und damit bei Bedarf leicht zu finden und zu ändern. Die Funktion `DokEigenschaft` wird weiter unten beschrieben.

Dann erzeugt die Sub `Etikett` ein neues Dokument, passt dessen Seitenlayout an und fügt ein Barcode-Feld ein. Nach einigen Absatzformatierungen druckt sie das neu erzeugte Dokument und schließt es. Zuletzt aktualisiert sie den Nummernstand in den Dokumenteneigenschaften und speichert das Etiketten-Tool, um den neuen Nummernstand zu sichern.

```
Const ETIKETTENDRUCKER As String = "Microsoft Print to PDF"
Const NUM_PROPNAME As String = "Inventarnummer"
Const NUM_INITIAL As Long = 100

Sub Etikett()
    Dim dok As Document
    Dim tool As Document
    Dim invNum As Long
    Dim barcodeDef As String

    Set tool = ThisDocument
    invNum = DokEigenschaft(tool.CustomDocumentProperties, _
        NUM_PROPNAME, NUM_INITIAL)

    Set dok = Documents.Add
    With dok.PageSetup
```

```

    .PageWidth = CentimetersToPoints(7)
    .PageHeight = CentimetersToPoints(5)
    .RightMargin = 2
    .LeftMargin = 2
    .TopMargin = 2
    .BottomMargin = 2
End With

barcodeDef = "DISPLAYBARCODE """ & invNum & """ QR \q 3"
dok.Fields.Add dok.Range(0, 0), wdFieldEmpty, barcodeDef, False
dok.Content.InsertAfter (vbCrLf & "Inventar " & invNum)
' vbCrLf = Zeilenumbruch

With dok.Content.Paragraphs.Format
    .Alignment = wdAlignParagraphCenter
    .SpaceAfter = 0
    .SpaceBefore = 0
End With

With Dialogs(wdDialogFilePrintSetup)
    .Printer = ETIKETTENDRUCKER
    .DoNotSetAsSysDefault = True
    .Execute
End With

dok.PrintOut
dok.Close False
tool.CustomDocumentProperties(NUM_PROPNAME).Value = invNum + 1
tool.Save
End Sub

```

Das Auslesen des aktuellen Nummernstands aus den Dokumenteneigenschaften leistet die Funktion `DokEigenschaft`, die dazu die Collection `CustomDocumentProperties`, den Namen der Dokumenteneigenschaft und den Initialwert als Parameter erhält. Falls die Dokumenteneigenschaft noch nicht existiert, legt die Funktion sie an und initialisiert sie. Die dabei verwendete Fehlerbehandlungsroutine ist in Abschn. 4.1.5 erläutert.

```

Function DokEigenschaft(props As DocumentProperties, _
    pname As String, initial As Long) As Long
Err.Clear
On Error Resume Next

DokEigenschaft = props(pname).Value
If Err.Number <> 0 Then

```

```
props.Add pname, False, msoPropertyTypeNumber, initial  
DokEigenschaft = initial  
Err.Clear  
End If  
End Function
```

Die Demo-Anwendung „Etiketten-Tool“ ist in der Wirtsanwendung Word entwickelt. Die Dokumenteigenschaften dienen ihr als Datenspeicher. Sie bietet jedoch keine benutzerfreundliche Bedienmöglichkeit, um ein Etikett zu generieren. Ihr Code lässt sich nur aus der Entwicklungsumgebung starten. Doch es gibt mehrere Optionen, um Makros in der Bedienoberfläche von Word zugänglich zu machen. Man kann etwa eine Schaltfläche zum Starten des Skripts in das Menüband integrieren. Das Vorgehen ist in Abschn. 9.3.3 für Outlook beschrieben. Auch die Symbolleiste für den Schnellzugriff kann ein Symbol für das Etiketten-Tool aufnehmen. Diese Möglichkeit ist in Abschn. 13.6 beschrieben.

Abschn. 13.3.3 erklärt, wie man erreicht, dass ein Skript beim Öffnen eines Dokuments automatisch startet. Ein solches Skript kann auch eine UserForm anzeigen, um weitergehende Eingabe- und Bedienmöglichkeiten für ein VBA-Tool auf Basis von Word zu schaffen. Abschn. 11.3.2 zeigt einen Weg, um ein Skript direkt aus dem Dateiverzeichnis auszuführen, ohne zuvor eine Wirtsanwendung zu öffnen.

6.7 Fazit

Der Blick in das Objektmodell von Word legt den Aufbau der damit erzeugten Dokumente offen und zeigt die Funktionsvielfalt dieser Software. Angesteuert durch VBA und integriert mit anderen Anwendungen zeigt sich Word als ein mächtiges Werkzeug, dessen Möglichkeiten über das manuelle Erfassen und Formatieren von Texten deutlich hinausgehen.

Literatur

1. „Sample Database | BIRT“. <https://eclipse.github.io/birt-website/docs/template-sample-database/> (zugegriffen 29. August 2022).
2. D. Rado, „Inserting text at a bookmark without deleting the bookmark“, *The Word MVP Site*, 2020. <https://wordmvp.com/FAQs/MacrosVBA/InsertingTextAtBookmark.htm> (zugegriffen 31. August 2022).
3. o365devx und olprod, „Selection-Objekt (Word)“, *Microsoft Docs*, 7. April 2022. <https://docs.microsoft.com/de-de/office/vba/api/word.selection> (zugegriffen 25. August 2022).
4. Microsoft, „Feldfunktionen: DisplayBarcode“, *Microsoft Support*. <https://support.microsoft.com/de-de/office/feldfunktionen-displaybarcode-6d81eadc-762d-4b44-ae81-f9d3d9e07be3?ui=de&rs=de-de&ad=de> (zugegriffen 23. August 2022).



PowerPoint

7

Inhaltsverzeichnis

7.1	Einführung in das Objektmodell von PowerPoint	118
7.1.1	Die Collection Shapes.....	118
7.1.2	Der Objekttyp Shape	118
7.1.3	Shape-Objekte bearbeiten	120
7.2	Demo-Anwendung „PowerPoint-Generator“	122
7.2.1	Präsentationsvorlage und Excel-Tool.....	123
7.2.2	VBA-Module	124
7.2.2.1	Modul Einstellungen	125
7.2.2.2	Modul Main.....	127
7.3	Fazit	129
	Literatur.....	129

PowerPoint ist ein mächtiges Werkzeug und kann sehr aufwendige, auch animierte Präsentationen realisieren. Zur Unterstützung von Arbeitsabläufen in Unternehmen tragen solche dynamischen Möglichkeiten jedoch eher wenig bei. Dieses Kapitel befasst sich damit, PowerPoint-Folien automatisiert mit Inhalten zu füllen. Dies spart Routinearbeit, verhindert Übertragungsfehler und bringt standardisierte Ergebnisse, wenn regelmäßig Präsentationen zu bestimmten Themen mit varierenden Daten benötigt werden.

Nach einem Überblick über relevante Teile des PowerPoint-Objektmodells und einigen kleinen Code-Beispielen präsentiert das Kapitel eine Demo-Anwendung, die automatisch Folien einer PowerPoint-Präsentation aktualisiert. Die Demo-Anwendung manipuliert vordefinierte Vorlagen anstatt PowerPoint-Präsentationen komplett durch VBA-Skripte zu erstellen. Eine solche Anwendungsarchitektur behandelt die wechselnden Inhalte getrennt von ihrer visuellen Darstellungsform, Design und Layout sind un-

abhängig von den anzuzeigenden Inhalten in Vorlagen definiert. Dies hat zwei wesentliche Vorteile: Erstens bleiben die VBA-Anteile der Anwendung schlank und überschaubar. Zweitens lässt sich die Optik der erzeugten Präsentationen manuell in den Vorlagen ändern, ohne in den VBA-Code einzugreifen.

Ein Schwerpunkt liegt hier darauf, eine konfigurierbare Demo-Anwendung zu gestalten: Einstellungen und Vorgaben sind auf einer Konfigurationsseite zugänglich und lassen sich dort anpassen. Die Demo-Anwendung liest die Einstellungen von dort ein und wendet sie an.

7.1 Einführung in das Objektmodell von PowerPoint

Auf den übergeordneten Ebenen ist das Objektmodell von PowerPoint ähnlich aufgebaut wie die Objektmodelle von Excel und Word. Den Einstieg bildet das Application-Objekt, das eine laufende Instanz der Anwendung PowerPoint repräsentiert. Dieses besitzt eine Collection Presentations, in der es die offenen Präsentationen verwaltet. Eine Präsentation hat den Objekttyp Presentation. Sie verwaltet ihre Folien in der Collection Slides. Für Folien bietet das Objektmodell den Objekttyp Slide.

7.1.1 Die Collection Shapes

Interessant ist, wie das Objektmodell die Inhalte einer Folie modelliert, denn eine Folie kann ganz unterschiedliche Arten von Inhalten zeigen wie Bilder, Diagramme, Texte, Tabellen und weitere. Das PowerPoint-Objektmodell verwendet für alle Arten von Inhalten denselben Objekttyp Shape, im Folgenden auch als Inhaltselement bezeichnet. Ein Slide-Objekt besitzt eine Collection Shapes, die alle seine Inhaltselemente als Objekte des Typs Shape speichert. Um ein Inhaltselement zu einer Folie hinzuzufügen, bietet die Collection Shapes verschiedene spezifische Methoden wie AddPicture, AddShape, AddTable, AddTitle, AddTextBox und weitere.

7.1.2 Der Objekttyp Shape

Der Objekttyp Shape repräsentiert unterschiedliche Arten von Inhalten. Ein Shape-Objekt besitzt ein Feld Type, welches den Typ des Inhaltselements angibt. Vom Typ hängt ab, welche Felder des Shape-Objekts gesetzt sind oder auch nicht. Ein Shape-Objekt hat Methoden, um dies abzufragen, wie der folgende Code-Schnipsel demonstriert. Er läuft in der VBA-Entwicklungsumgebung von PowerPoint und bezieht sich auf Abb. 7.1. Diese zeigt eine PowerPoint-Folie mit vier Inhaltselementen, von denen eines selektiert ist. Die Namen der Inhaltselemente sind im Auswahlbereich rechts von der Folie zu sehen.

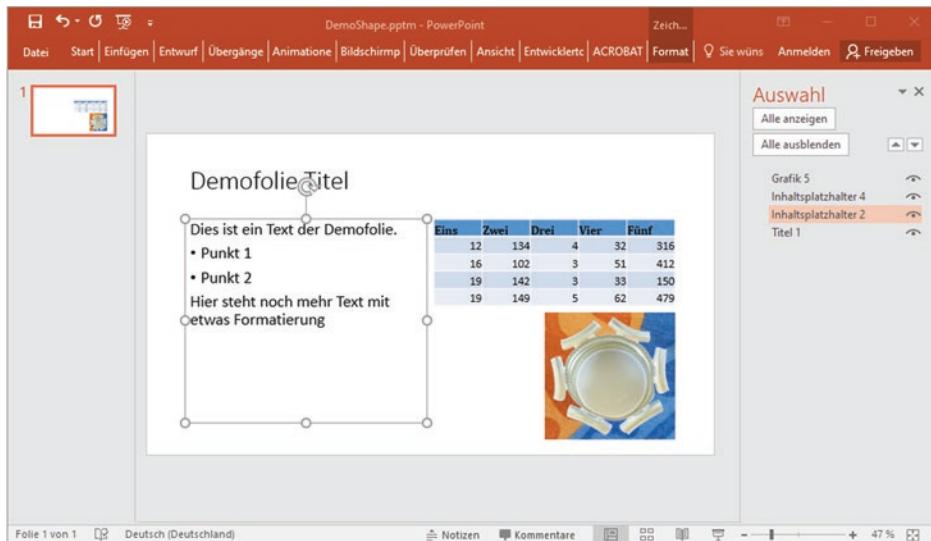


Abb. 7.1 PowerPoint-Vorlage für Code-Beispiele und Demo-Anwendung

```
Sub ShapesAuflisten()
Dim elem As Shape
For Each elem In ActivePresentation.Slides(1).Shapes
    Debug.Print
    Debug.Print "**** " & elem.Name
    Debug.Print "ist Platzhalter? " & (elem.Type = msoPlaceholder)
    Debug.Print "ist Bild? " & (elem.Type = msoPicture)
    Debug.Print "hat Text? " & elem.HasTextFrame
    Debug.Print "hat Tabelle? " & elem.ToTable
    Debug.Print "hat Diagramm? " & elem.HasChart
Next elem
End Sub
```

Die geklammerten Ausdrücke `(elem.Type = msoPlaceholder)` und `(elem.Type = msoPicture)` sind Vergleiche und liefern als Vergleichsergebnis einen Booleschen Wert True oder False. Der Verketten-Operator `&` verwandelt den Booleschen Wert in einen String und fügt ihn an einen anderen String an, der dann mit dem Befehl `Debug.Print` ausgegeben wird. Für die Folie in Abb. 7.1 erzeugt der Code-Schnipsel diese Ausgabe:

```
*** Title 1
ist Platzhalter? Wahr
ist Bild? Falsch
hat Text? Wahr
```

```
hat Tabelle? Falsch
hat Diagramm? Falsch

*** Content Placeholder 2
ist Platzhalter? Wahr
ist Bild? Falsch
hat Text? Wahr
hat Tabelle? Falsch
hat Diagramm? Falsch

*** Content Placeholder 4
ist Platzhalter? Wahr
ist Bild? Falsch
hat Text? Falsch
hat Tabelle? Wahr
hat Diagramm? Falsch

*** Picture 5
ist Platzhalter? Falsch
ist Bild? Wahr
hat Text? Falsch
hat Tabelle? Falsch
hat Diagramm? Falsch
```

Wie der Code-Schnipsel zeigt, besitzt jedes Shape-Objekt ein Feld `Name`. Außerdem besitzt es die Felder `Top`, `Left`, `Height` und `Width`, die seine Position und Größe angeben.

Die folgenden Demos greifen auf Inhaltselemente zu und ändern sie. Auch sie beziehen sich auf die Folie in Abb. 7.1. Um auf ein bestimmtes Shape zuzugreifen, kann man es einfach mit seinem Namen identifizieren, den der Auswahlbereich anzeigt. Dies funktioniert mit den Namen, die PowerPoint beim Einfügen eines Inhaltselements automatisch vergibt, und auch mit Namen, die manuell im Auswahlbereich oder mit VBA-Befehlen zugewiesen wurden. Die folgenden Demo-Prozeduren zeigen auch, dass Text-Inhaltselemente und Tabellen in PowerPoint ähnlich aufgebaut sind wie Texte und Tabellen in Word.

7.1.3 Shape-Objekte bearbeiten

Dieses Demo-Skript analysiert und modifiziert ein Textelement:

```
Sub TextelementAnsprechen()
Dim elem As Shape
```

```
Set elem = ActivePresentation.Slides(1) _
.Shapes("Inhaltsplatzhalter 2")

If elem.HasTextFrame Then
    If elem.TextFrame.HasText Then
        With elem.TextFrame.TextRange
            Debug.Print Left(.Text, 10) & "..."      ' Dies ist e...
            Debug.Print .Length                      ' 97
            Debug.Print .Words.Count                 ' 19
            .Words(19).Font.Size = 44
            .Words(19).Font.Name = "Times New Roman"
            .Words(19).Font.Italic = msoTrue
            .Words(19).Font.Color.RGB = RGB(200, 80, 80)
        End With
    End If
End If
End Sub
```

Die Zellen eines Tabellen-Elements speichern ihre Inhalte ebenfalls als Shape-Objekte.
Das folgende Skript greift darauf zu:

```
Sub TabelleAnsprechen()
Dim elem As Shape
Dim tabelle As Table
Set elem = ActivePresentation.Slides(1) _
.Shapes("Inhaltsplatzhalter 4")

If elem.ToTable Then
    Set tabelle = elem.Table
    tabelle.Cell(2, 2).Shape.TextFrame2.TextRange.text = "999"
    tabelle.Cell(2, 2).Shape.TextFrame2.TextRange.Font.Size = "32"
    tabelle.Cell(2, 2).Shape.Fill.ForeColor.RGB = RGB(200, 0, 200)
End If
End Sub
```

Die folgende Demo-Prozedur manipuliert ein Shape vom Typ msоСPicture, also ein Bild.

```
Sub GrafikAnsprechen()
Dim elem As Shape
Set elem = ActivePresentation.Slides(1).Shapes("Grafik 5")
```

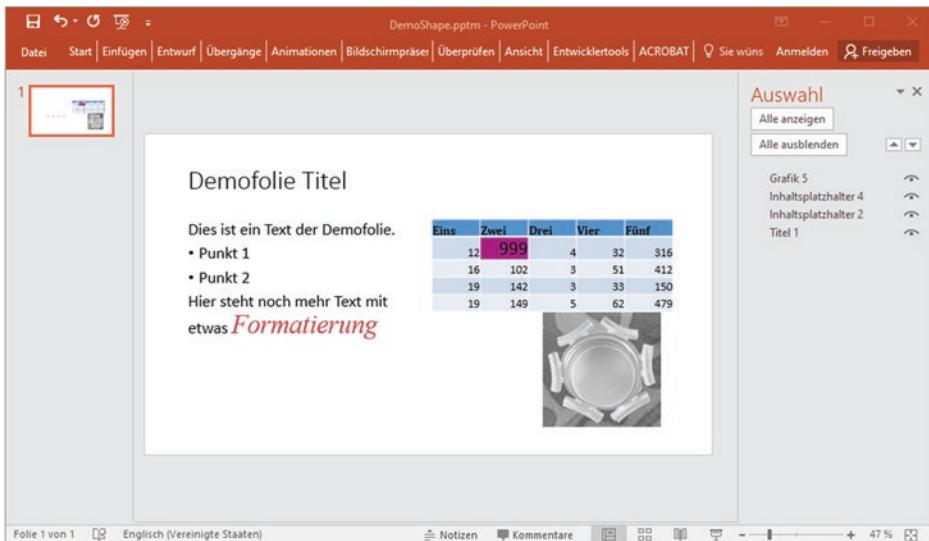


Abb. 7.2 PowerPoint-Vorlage nach Änderungen durch VBA-Befehle

```

elem.Height = 200
elem.Width = 200
If elem.Type = msoPicture Then
    elem.PictureFormat.ColorType = msoPictureGrayscale
End If
End Sub

```

Nach der Ausführung dieser Prozeduren sieht die Folie von Abb. 7.1 aus wie in Abb. 7.2.

7.2 Demo-Anwendung „PowerPoint-Generator“

Ein Unternehmen erstellt Analysen und sammelt die Ergebnisdaten in einem Excel-Arbeitsblatt. Der Kunde erhält das Analyseergebnis in einer PowerPoint-Präsentation aufbereitet zu einem Diagramm mit Titel und begleitendem Text. Die Demo-Anwendung „PowerPoint-Generator“ unterstützt dabei, indem sie die PowerPoint-Präsentation zu den Daten automatisch generiert.

Der PowerPoint-Generator ist in Excel erstellt. Abb. 7.3 zeigt die Verweise auf die benötigten Objektbibliotheken von PowerPoint und der Microsoft Scripting Runtime. Zum PowerPoint-Generator gehört außerdem eine Muster-PowerPoint-Präsentation mit formatierten Platzhaltern für das Diagramm, den Titel und den Text. Der PowerPoint-Generator öffnet die Musterpräsentation, fügt die Inhaltselemente ein und speichert die Präsentation unter neuem Namen in ein Zielverzeichnis.

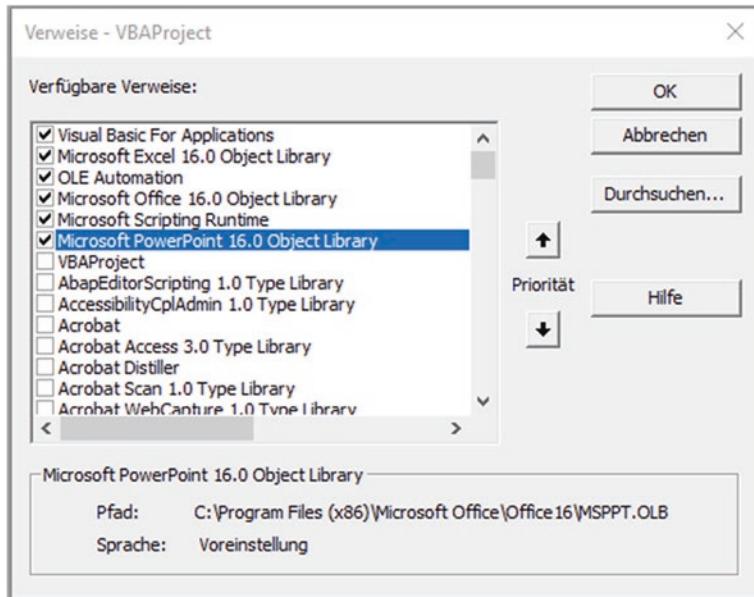


Abb. 7.3 Verweise für die PowerPoint-Demo-Anwendung

7.2.1 Präsentationsvorlage und Excel-Tool

Die Muster-PowerPoint-Präsentation ist in Abb. 7.4 zu sehen. Der Auswahlbereich ist offen, damit die Bezeichner der Inhaltselemente zu sehen sind. Der VBA-Code greift mit den Bezeichnern auf die Inhaltselemente zu.

Die Excel-Arbeitsmappe des PowerPoint-Generators besteht aus vier Arbeitsblättern:

- Auf dem Arbeitsblatt „Eingabe“ findet der PowerPoint-Generator die aufzubereitenden Daten, den Titel und den Text, siehe Abb. 7.5.
- Das Arbeitsblatt „Diagrammvorlage“ enthält ein mit Dummy-Daten erstelltes, fertig formatiertes Diagramm. Dieses Arbeitsblatt zeigt Abb. 7.6. Der PowerPoint-Generator befüllt es mit den aufzubereitenden Daten und kopiert das Diagramm in die PowerPoint-Präsentation.
- Auf dem Arbeitsblatt „Einstellungen“ sind die Pfade der Vorlagen und die Position der Daten hinterlegt, vergleiche Abb. 7.7. Da die Einstellungen hier sichtbar und zugänglich sind, lässt sich der PowerPoint-Generator flexibel für andere User und Anwendungssituationen konfigurieren, ohne in den Code einzugreifen.
- Das Arbeitsblatt „Dokumentation“, das nicht abgebildet ist, enthält Texte, die den PowerPoint-Generator beschreiben und die Bedienung erklären.



Abb. 7.4 Muster-PowerPoint-Präsentation für die Demo-Anwendung „PowerPoint-Generator“

F22								
A	B	C	D	E	F	G	H	I
1 Daten und Texte für die Präsentation hier eingeben								
2								
3 Daten								
4 Name	Baseline	Nina	Chris	Oma	Opa	Papa		
5 Entwurf	100000	56789	12345	54322	23444	90000		
6 Produktion	100000	50000	45545	20000	52525	80000		
7 Transport & Verpackun	100000	60000	80000	10000	10000	56666		
8 Kosten - Gesamt	100000	70000	80000	30000	10000	11111		
9								
10 Texte								
11 Titel der Folie								
12 Informativer Text								
13								
Eingabe Dokumentation Einstellungen Diagrammvorlage								

Abb. 7.5 Excel-Arbeitsblatt „Eingabe“ der Demo-Anwendung „PowerPoint-Generator“

7.2.2 VBA-Module

Der VBA-Code verteilt sich auf zwei Module. Das Modul `Einstellungen` ist dafür zuständig, den PowerPoint-Generator gemäß den Angaben auf dem Arbeitsblatt „Einstellungen“ zu konfigurieren. Das Modul `Main` enthält den Code, der die Präsentation erzeugt.

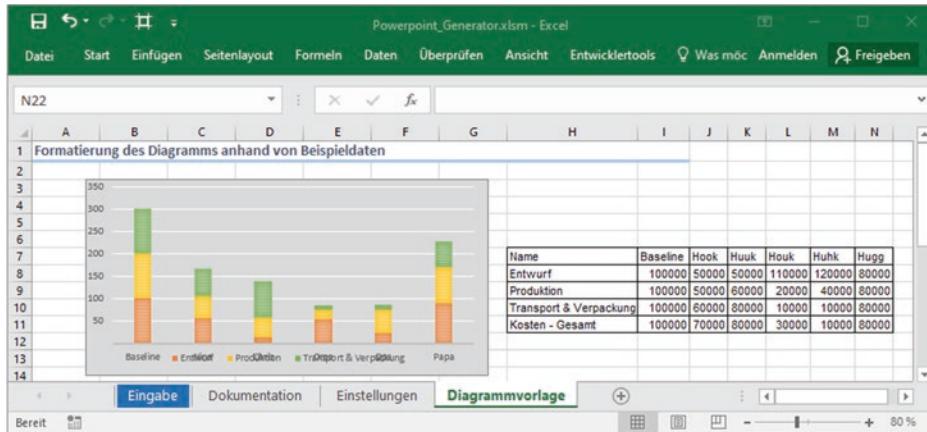


Abb. 7.6 Excel-Arbeitsblatt „Diagrammvorlage“ der Demo-Anwendung „PowerPoint-Generator“

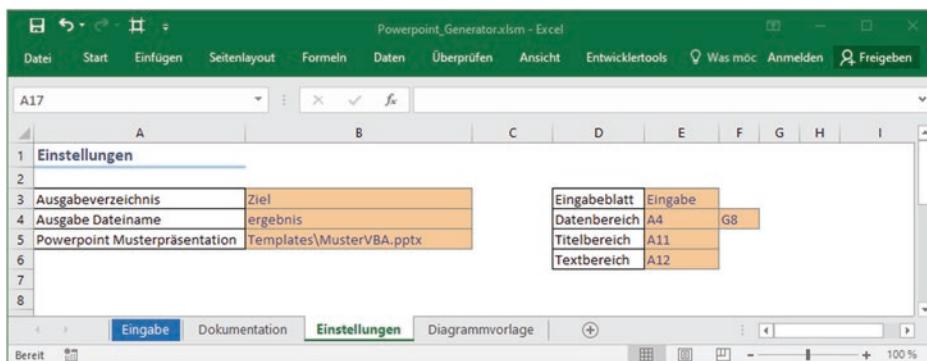


Abb. 7.7 Excel-Arbeitsblatt „Einstellungen“ der Demo-Anwendung „PowerPoint-Generator“

7.2.2.1 Modul Einstellungen

Das Modul Einstellungen definiert zwei Datentypen EingabeT und EinstellungT und zwei Funktionen. Die Funktion Eingabe überträgt die auf dem Arbeitsblatt „Einstellungen“ hinterlegten Werte in eine Variable des Typs EingabeT.

```
Type EingabeT
    ausgabeverz As String
    ausgabenname As String
    musterpfad As String
    eingabenname As String
    datenadresse As String
    titelzelle As String
```

```
    textzelle As String
End Type

Function Eingabe() As EingabeT
Dim eingeb As EingabeT

With Worksheets("Einstellungen")
    eingeb.ausgabeverz = .Cells(3, 2).Value
    eingeb.ausgabename = .Cells(4, 2).Value
    eingeb.musterpfad = .Cells(5, 2).Value
    eingeb.eingabename = .Cells(3, 5).Value
    eingeb.datenadresse = .Cells(4, 5).Value & ":" & .Cells(4, 6).Value
    eingeb.titelzelle = .Cells(5, 5).Value
    eingeb.textzelle = .Cells(6, 5).Value
End With
Eingabe = eingeb
End Function
```

Die Funktion `Einstellung` baut daraus die benötigten Dateinamen, Pfade und Datenobjekte und speichert diese in eine Variable des Typs `EinstellungT`. Sie richtet auch das Zielverzeichnis ein, wenn es noch nicht existiert, und generiert einen möglichst eindeutigen Dateinamen für die neu erzeugte PowerPoint-Präsentation.

```
Type EinstellungT
    ausgabedat As String
    daten As Range
    titelbereich As Range
    textbereich As Range
    diagramm As Chart
    musterpfad As String
End Type

Function Einstellung(eing As EingabeT) As EinstellungT
Dim einst As EinstellungT
Dim ausgabepfad As String
Dim adresse As String

With eing
    ausgabepfad = ThisWorkbook.Path & "\" & .ausgabeverz
    If Not FSO.FolderExists(ausgabepfad) Then
        FSO.CreateFolder (ausgabepfad)
    End If
    einst.ausgabedat = ausgabepfad & "\"
    & Format(Now(), "YY_MM_DD_hh_nn_ss") & "_" & .ausgabename
End Function
```

```

einst.musterpfad = ThisWorkbook.Path & "\" & .musterpfad
adresse = .eingabename & "!" & .titelzelle & ":" & .titelzelle
Set einst.titelbereich = Range(adresse)

adresse = .eingabename & "!" & .textzelle & ":" & .textzelle
Set einst.textbereich = Range(adresse)

adresse = .eingabename & "!" & eing.datenadresse
Set einst.daten = Range(adresse)
End With

Set einst.diagramm = Worksheets("Diagrammvorlage") _
.Shapes("Musterdiagramm").Chart
Einstellung = einst
End Function

```

In diesen beiden Funktionen können Laufzeitfehler auftreten. Diese behandelt die aufrufende Prozedur im Modul Main.

7.2.2.2 Modul Main

Das Modul Main enthält die Prozedur Main und die Prozedur PraesentationErzeugen. Hier sind auch zwei Variablen auf Modulebene deklariert sowie eine Public-Variable für das FileSystemObject, das in beiden Modulen verwendet wird.

Die Prozedur Main wird mit der Schaltfläche gestartet, die in Abb. 7.5 zu sehen ist. Die Prozedur erzeugt ein FileSystemObject und eine PowerPoint-Application. Sie konfiguriert den PowerPoint-Generator mithilfe der beiden Funktionen Eingabe und Einstellung, öffnet die Musterpräsentation und führt dann die Prozedur PraesentationErzeugen aus. Die Prozedur Main erledigt auch die Fehlerbehandlung. Im Erfolgsfall wie im Fehlerfall schließt die Prozedur Main nicht mehr benötigte Anwendungen und gibt den Speicherplatz der verwendeten Objekte frei:

```

Public FSO As FileSystemObject
Private PPApp As PowerPoint.Application
Private musterpraes As PowerPoint.Presentation

Sub Main()
Dim einstell As EinstellungT

On Error GoTo Fehler
Set FSO = CreateObject("Scripting.FileSystemObject")
Set PPApp = CreateObject("Powerpoint.Application")
einstell = Einstellung(Eingabe())
Set musterpraes = PPApp.Presentations.Open(einstell.musterpfad)

```

```
PraesentationErzeugen einstell
MsgBox ("Präsentation erzeugt.")

GoTo Aufraeumen

Fehler:
Select Case Err.Number
Case 1004
    MsgBox (Err.Number & ":" & Err.Description & Chr(13) _
        & "Bitte Einstellungen prüfen.")
Case 1008
    MsgBox (Err.Number & ":" & Err.Description & Chr(13) _
        & "Das Ausgabeverzeichnis für " & einst.ausgabedat & _
        " lässt sich nicht einrichten. Bitte Einstellungen ändern.")
Case Is < -200000
    MsgBox (Err.Number & ":" & Err.Description & Chr(13) _
        & "Sind Verzeichnis und Musterpräsentation " _
        & einst.musterpfad & " vorhanden?" & Chr(13) _
        & "Die Musterpräsentation darf nicht offen sein.")
Case Else
    MsgBox (Err.Number & ":" & Err.Description & Chr(13) _
        & "Unvorhergesehener Fehler.")
End Select
```

Aufraeumen:

```
On Error Resume Next
musterpraes.Close
Set musterpraes = Nothing

If PPApp.Presentations.Count = 0 Then
    PPApp.Quit
    Set PPApp = Nothing
End If
Set FSO = Nothing
MsgBox ("Programm beendet.")
End Sub
```

Die Prozedur PraesentationErzeugen erstellt eine PowerPoint-Präsentation, indem sie Titel und Text aus der Excel-Arbeitsmappe liest und in die entsprechenden PowerPoint-Inhaltsplatzhalter einfügt. Das Diagramm (Chart) wird noch in Excel mit Daten vom Arbeitsblatt „Eingabe“ befüllt und dann per Copy&Paste in die PowerPoint-Präsentation übertragen:

```
Sub PraesentationErzeugen(einst As Einstellung)
Dim folie As PowerPoint.Slide
Dim diagrammPlatzhalter As PowerPoint.Shape

Set folie = musterpraes.Slides(1)

With folie.Shapes("Titelplatzhalter")
    .TextFrame.TextRange.Text = einst.titelbereich
End With

With folie.Shapes("Textplatzhalter")
    .TextFrame.TextRange.Text = einst.textbereich
End With

Set diagrammPlatzhalter = folie.Shapes("Diagrammplatzhalter")
einst.diagramm.SetSourceData Source:=einst.daten
einst.diagramm.ChartArea.Copy
diagrammPlatzhalter.Select msoTrue
musterpraes.Windows(1).View.PasteSpecial (ppPasteShape)

musterpraes.SaveCopyAs (einst.ausgabedat)
musterpraes.ExportAsFixedFormat einst.ausgabedat _
    & ".pdf", ppFixedFormatTypePDF, ppFixedFormatIntentPrint
End Sub
```

7.3 Fazit

Im Vergleich zu anderen Office-Anwendungen ist VBA in PowerPoint nur mit einigen Einschränkungen nutzbar. So gibt es keinen Makrorecorder, um Skripte als Ausgangsbasis für eigenen Code aufzuzeichnen, und wenn der VBA-Code auf PowerPoint-Ereignisse reagieren soll, muss dies erst eingerichtet werden [1]. Dennoch lassen sich auch für PowerPoint wirksame Helfer und Automatisierungen mit VBA realisieren. Beispielsweise können Makros automatisch Gliederungsfolien als Inhaltsverzeichnisse für umfangreiche Präsentationen erzeugen und auch Hyperlinks zu einzelnen Seiten einfügen. Die Integration mit anderen Anwendungssystemen erschließt weitere interessante Möglichkeiten.

Literatur

1. o365devx, A. Jerabek, K. Brandl, Office GSX, und L. Caputo, „Verwenden von Ereignissen mit dem Application-Objekt“, *VBA-Referenz für Office/PowerPoint*, 19. Februar 2023. <https://learn.microsoft.com/de-de/office/vba/powerpoint/how-to/use-events-with-the-application-object> (zugriffen 27. Februar 2023).



Outlook I – Elemente automatisch bearbeiten

8

Inhaltsverzeichnis

8.1	Anmeldung in Outlook	132
8.2	Elemente des Outlook-Objektmodells I: Ordner	133
8.2.1	Die Objekttypen Folder und Folders	133
8.2.2	Application-Objekt und NameSpace-Objekt	133
8.2.3	Code-Beispiel: Zugriff auf Outlook-Ordner	134
8.2.4	Code-Beispiel: Navigation durch die Ordnerstruktur mit Parent und Folders	136
8.2.5	Code-Beispiel: Direkter Zugriff auf Ordner mit der EntryID	136
8.3	Objekttypen für Elemente	138
8.3.1	Code-Beispiel: Elemente in einem Outlook-Ordner auflisten	138
8.3.2	Nice to know: Unterschiedliche Objekttypen in einer Collection und type cast	139
8.4	Demo-Anwendung „Posteingangsverwaltung“	139
8.5	Demo-Anwendung „Anhängeverwaltung“ mit komplexen Filtern	141
8.5.1	Komplexe DASL-Filter	142
8.5.1.1	DASL-Filter und JET-Filter	142
8.5.1.2	Komplexe Filter definieren	143
8.5.1.3	Nice to know: Anführungszeichen im String „escapen“	143
8.5.2	Berechnung von Datumsangaben	144
8.5.3	Der Code der Demo-Anwendung	145
8.6	Demo-Anwendung „Termine-Controlling“	146
8.7	Demo-Anwendung „Mailing-Tool“	148
8.7.1	Eine einfache E-Mail erstellen und versenden	148
8.7.2	Formatierte E-Mails versenden	149
8.8	Fazit und Ausblick	152
	Literatur	152

Outlook ist Bestandteil vieler Arbeitsabläufe und bietet damit großes Potenzial, durch Automatisierungen die Effizienz und Qualität von Arbeit zu steigern. Dieses Kapitel befasst sich damit, E-Mails, Kalendereinträge und andere Outlook-Inhalte automatisiert in größerer Anzahl zu bearbeiten:

- Eingegangene E-Mails des aktuellen Tags auflisten und Besprechungseinladungen beantworten
- Mailanhänge automatisch herunterladen und in ein Dateiverzeichnis speichern
- Meetings zu einem bestimmten Thema in Excel auswerten
- Individualisierte Mailings für eine Liste von Adressaten erzeugen und versenden.

Die Code-Beispiele dieses Kapitels laufen in Excel als Wirtsanwendung. Deshalb wird zunächst erklärt, wie ein VBA-Skript aus Excel auf die Anwendung Outlook zugreifen und Zugang zu Daten erhalten kann. Darauf folgt eine Einführung in das Objektmodell von Outlook mit Fokus auf die Objekttypen `Folder` und `Item`. Die Code-Beispiele greifen auf Outlook-Ordner zu und selektieren und bearbeiten die darin enthaltenen Elemente. Eine Demo-Anwendung zeigt, wie sich komplexe Filter in der Bedienoberfläche von Outlook erstellen und dann in den VBA-Code übernehmen lassen. Eine weitere Demo-Anwendung realisiert ein Mailing-Tool für den Versand von E-Mails an viele Empfänger aus Excel.

8.1 Anmeldung in Outlook

Wie die bisher betrachteten Office-Anwendungen läuft auch Outlook als Anwendung auf dem Computer eines Benutzers. Anders als Excel, Word und PowerPoint arbeitet Outlook gewöhnlich in Verbindung mit einem Microsoft Exchange-Server, der E-Mails und sonstige Daten von vielen Benutzern verwaltet und verteilt. Outlook übernimmt dabei die Rolle des Clients (siehe dazu auch Kap. 10). Damit der Exchange-Server die richtigen Daten zur Verfügung stellen kann, müssen Benutzer und auch ein VBA-Skript mit dem passenden Benutzerprofil am Server angemeldet sein.

Den Zugang zu den Daten eines Benutzers regelt im Outlook-Objektmodell das `NameSpace`-Objekt. Dieses bietet eine `Logon`-Methode, mit der man ein Benutzerprofil mit Kennung und Passwort anmelden kann. Diese Methode wird aber nur in sehr seltenen Fällen gebraucht, denn die Anmeldung lässt sich mit anderen Mechanismen eleganter erreichen, und vor allem auch, ohne ein Klartext-Passwort im VBA-Skript zu hinterlegen. Der erste Mechanismus ist die automatische Anmeldung: Wenn Outlook startet, meldet es den aktuellen Benutzer selbstständig und unbemerkt an. Es verwendet dabei das Standard-Benutzerprofil, das gewöhnlich für jeden Benutzer konfiguriert ist. Auch ein VBA-Skript kann die automatische Anmeldung auslösen [1].

Ein zweiter Mechanismus nutzt eine Besonderheit von Outlook. Zu jeder Zeit kann nämlich nur eine einzige Instanz der Anwendung Outlook auf einem Rechner laufen.

Auch wenn mehrere Outlook-Fenster offen sind, gehören diese immer zu derselben Outlook-Instanz und laufen mit demselben Benutzerprofil [1]. Wenn Outlook auf einem Rechner bereits läuft, kann ein Skript daher keine zusätzliche Outlook-Instanz starten. Es klinkt sich stattdessen automatisch bei der bereits laufenden Outlook-Instanz ein und übernimmt auch das dort aktive Benutzerprofil. Eine explizite Anmeldung entfällt. Diese Anmelde-Mechanismen werden in den Code-Beispielen eingesetzt.

8.2 Elemente des Outlook-Objektmodells I: Ordner

Ein Skript, das Arbeiten in Outlook automatisiert, greift dazu auf Elemente des jeweiligen Benutzers zu. Die Elemente, also E-Mail-Nachrichten, Termine, Kontakte etc., liegen in verschiedenen Ordner. Die erste Herausforderung bei der Skriptentwicklung besteht darin, auf die interessierenden Ordner zuzugreifen.

8.2.1 Die Objekttypen Folder und Folders

Outlook organisiert Ordner in Collections des Typs `Folders`. Ein `Folders`-Objekt besitzt die typischen Eigenschaften und Methoden einer Collection wie `Count`, `Add`, `Remove`, `Item` etc. Seine Elemente haben den Objekttyp `Folder`. Dieser Teil des Outlook-Objektmodells ist rekursiv aufgebaut, so wie auch das in Kap. 5 betrachtete Dateiverzeichnissystem:

- Ein Objekt des Typs `Folder` besitzt selbst wieder eine Collection `Folders`, die seine Unterordner enthält.
- Jedes `Folder`-Objekt und jedes `Folders`-Objekt besitzt ein Feld `Parent`, das auf das ihm übergeordnete `Folders`-Objekt verweist.

Ein `Folder`-Objekt besitzt außerdem eine Collection `Items`, die seine Elemente verwaltet. Mehr zu `Items` und Elementen folgt in Abschn. 8.3.

In Outlook gibt es vordefinierte Ordner wie Posteingang und Kalender, die für jeden Outlook-Nutzer standardmäßig eingerichtet sind, die sogenannten Standard-Ordner. Benutzer können weitere neben- oder untergeordnete Ordner anlegen und mit eigenen Bezeichnern frei benennen. Weiterhin besitzen Benutzer Zugriffsrechte für Ordner, die andere Benutzer freigegeben haben, und für Ordner, die auf dem Exchange-Server veröffentlicht sind.

8.2.2 Application-Objekt und NameSpace-Objekt

Das TopLevel-Element des Outlook-Objektmodells ist das `Application`-Objekt. Es verwaltet Outlook-Ordner nicht direkt, sondern mittels eines `NameSpace`-Objekts. Ein

NameSpace-Objekt erhält man mit der Methode GetNamespace des Application-Objekts. Diese wird mit dem Parameter MAPI aufgerufen. MAPI steht für „Messaging Application Program Interface“ und ist der einzige mögliche Parameter für GetNamespace.

Das NameSpace-Objekt bietet drei Methoden, um auf Ordner zuzugreifen:

- Die Methode GetDefaultFolder bietet Zugang zu den eigenen Ordner eines Benutzers und zu Ordner, die auf dem Exchange-Server veröffentlicht sind.
- Die Methode GetSharedDefaultFolder bietet Zugriff auf Ordner, die andere Benutzer freigegeben haben.
- Die Methode GetFolderFromID liefert beliebige Ordner, deren EntryID bekannt ist.

Für alle Zugriffsmethoden muss das NameSpace-Objekt initialisiert sein. Es wird initialisiert, indem man auf einen Standard-Ordner zugreift. Standard-Ordner werden durch Outlook-Konstanten identifiziert, siehe einige davon in Tab. 8.1. Für den Zugriff auf geteilte Ordner eines anderen Benutzers gibt man diesen mit der E-Mail-Adresse oder dem Anzeigennamen an.

8.2.3 Code-Beispiel: Zugriff auf Outlook-Ordner

Der folgende Code-Schnipsel zeigt den Zugriff auf verschiedene Standard-Ordner und ihre Unterordner. Die Outlook-Objektbibliothek wird referenziert wie in Abb. 8.1. Da der Code-Schnipsel in Excel entwickelt ist, instanziert er zuerst ein Application-Objekt für Outlook. Dann steuert er folgende Ordner an: (1) den Posteingang des angemeldeten Benutzers, (2) einen geteilten Kalender eines anderen Benutzers und (3) einen Ordner, der auf einem Exchange-Server veröffentlicht ist:

Tab. 8.1 Outlook-Konstanten für Standard-Outlook-Ordner [2]

Name	Wert	Beschreibung
olFolderCalendar	9	Kalender
olFolderContacts	10	Kontakte
olFolderInbox	6	Posteingang
olFolderSentMail	5	Gesendete Elemente
olPublicFoldersAllPublicFolders	18	Unterverzeichnis „Alle öffentlichen Ordner“ unter den auf Exchange veröffentlichten Ordnern. Nur für Exchange-Konten verfügbar

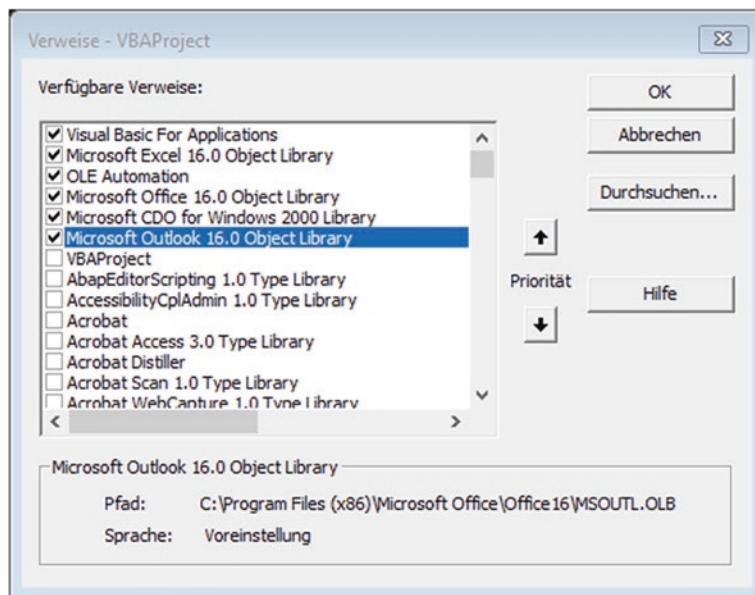


Abb. 8.1 Einbinden der Outlook-Objektbibliothek

```

Sub AufOrdnerZugreifen()
Dim olApp As New Outlook.Application
Dim nsp As Outlook.Namespace
Dim ordner As Outlook.Folder
Dim besitzer As Outlook.Recipient

Set nsp = olApp.GetNamespace("MAPI")

Set ordner = nsp.GetDefaultFolder(olFolderInbox)           ' (1)
Debug.Print ordner.Items.Count

Set besitzer = nsp.CreateRecipient("Anna Beta")
Set ordner = nsp.GetSharedDefaultFolder(besitzer, olFolderCalendar) _
    .Folders("Termine")
Debug.Print ordner.Name & ordner.Items.Count           ' (2)

Set ordner = _
    nsp.GetDefaultFolder(olPublicFoldersAllPublicFolders) _ 
    .Folders("Meine Organisation").Folders("Termine")      ' (3)
Debug.Print ordner.Name & ordner.Items.Count
End Sub

```

Der Code-Schnipsel gibt die Anzahl der Elemente im betreffenden Ordner aus. Die Bezeichner „Termine“, „Anna Beta“ und „Meine Organisation“ müssen durch Bezeichner ersetzt werden, die im jeweiligen System existieren.

8.2.4 Code-Beispiel: Navigation durch die Ordnerstruktur mit Parent und Folders

Der folgende Code-Schnipsel navigiert mit den Feldern `Folders` und `Parent` eines `Folder`-Objekts durch die Ordnerstruktur und erreicht so auch Ordner, die keinem Standard-Ordner untergeordnet sind:

```
Sub OrdnerNavigieren()
Dim olApp As New Outlook.Application
Dim posteingang As Outlook.Folder
Dim ordner As Outlook.Folder
Set posteingang = olApp.Session.GetDefaultFolder(olFolderInbox)

With posteingang.Parent
Debug.Print "Überordner: " & .Name
For Each ordner In .Folders
    Debug.Print .Name & ":" & ordner.Name
Next ordner
End With

For Each ordner In posteingang.Folders
    Debug.Print posteingang.Name & ":" & ordner.Name
Next ordner
End Sub
```

Die Navigation startet bei einem Standard-Ordner, nämlich dem Posteingang. Der Aufruf der Methode `GetDefaultFolder` (oder gegebenenfalls der Methode `GetSharedDefaultFolder`) mit einem Standard-Ordner initialisiert das `NameSpace`-Objekt. Das Skript durchläuft dann den übergeordneten Ordner und den Posteingang selbst und gibt die Bezeichner der darin enthaltenen Ordner aus.

8.2.5 Code-Beispiel: Direkter Zugriff auf Ordner mit der EntryID

Jeder Ordner hat eine `EntryID`. Ein Ordner lässt sich damit direkt ansteuern, ohne durch Ordnerstrukturen zu navigieren. Die `Sub EntryIDZeigen` des folgenden

Demo-Skripts öffnet den Outlook-Dialog **Pickfolder**, der die gesamte Ordnerstruktur zeigt. Hier kann man einen Ordner anwählen. Die Sub gibt die EntryID des gewählten Ordners aus und schließt den Dialog wieder.

```
Sub EntryIDZeigen()
Dim olApp As Object
Dim ordner As Outlook.Folder
Dim namensraum As Outlook.Namespace

Set olApp = CreateObject("Outlook.Application")
Set namensraum = olApp.GetNamespace("MAPI")
Set ordner = namensraum.PickFolder ' Ordner im Outlook-Dialog wählen
Debug.Print ordner.EntryID

olApp.Quit
Set olApp = Nothing
End Sub
```

Die Sub PerEntryIDZugreifen spricht den Ordner über seine EntryID an. Vorher muss sie die passende GetDefaultFolder-Methode ausführen, um das MAPI-Objekt zu initialisieren, hier für den Zugriff auf einen Ordner, der auf dem Exchange-Server veröffentlicht ist.

```
Sub PerEntryIDZugreifen()
Dim olApp As Outlook.Application
Dim ordner As Outlook.Folder
Dim namensraum As Outlook.Namespace

Set olApp = CreateObject("Outlook.Application")
Set namensraum = olApp.GetNamespace("MAPI")
Set ordner = olApp.GetNamespace("MAPI").GetDefaultFolder( _
    olPublicFoldersAllPublicFolders)      ' geeignet initialisieren

Set ordner = olApp.GetNamespace("MAPI").GetFolderFromID("00...000")
Debug.Print ordner.Name

Set olApp = Nothing
End Sub
```

Statt "00...000" setzt man die EntryID eines Ordners ein.

8.3 Objekttypen für Elemente

Das Outlook-Objektmodell enthält Objekttypen für die verschiedenen Arten von Elementen wie MailItem, AppointmentItem, MeetingItem, ContactItem, TaskItem und weitere. Bestimmte Felder und Methoden sind dabei in jedem Element-Objekttyp vorhanden. Eines davon ist das Feld Class, das den Objekttyp des Elements als Zahlencode angibt. Einige der Zahlencodes und ihre Outlook-Konstanten sind in Tab. 8.2 zusammengestellt. Die Collection Items eines Folder-Objekts kann einen Mix unterschiedlicher Elementtypen enthalten. Durch Abfrage ihrer Class-Eigenschaft kann ein VBA-Skript sie unterscheiden und geeignet behandeln, wie etwa im folgenden Code-Beispiel.

8.3.1 Code-Beispiel: Elemente in einem Outlook-Ordner auflisten

Der Code durchläuft den Posteingang mit einer ForEach-Schleife. Dabei verwendet er eine Schleifenvariable des Typs Object, da sie Objekte mit beliebigem Typ aufnehmen kann. Innerhalb der Schleife fragt der Code die Class-Eigenschaft jedes Elements ab. Elemente mit dem Wert olMail oder olMeetingRequest weist er einer Variablen mit dem passenden speziellen Objekttyp zu, also MailItem oder MeetingItem. Dann gibt er den Absender der Nachricht bzw. Meeting-Einladung aus. Elemente mit anderem Typ ignoriert er.

```
Sub ItemsAuflisten()
Dim outlApp As New Outlook.Application
Dim itm As Object
Dim inbox As Items
Dim meetreq As MeetingItem
Dim mailitm As MailItem
Dim max As Integer
max = 20 ' Stopper für volle Inboxes
```

Tab. 8.2 Outlook-Konstanten für Element-Objekttypen [3]

Outlook-Konstante	Nr	Bedeutung
olAppointment	26	AppointmentItem
olContact	40	ContactItem
olMail	43	MailItem
olTask	48	TaskItem
olMeetingRequest	53	MeetingItem, eine Anfrage
olMeetingResponsePositive	56	MeetingItem, eine Zusage

```
Set inbox = _
    outApp.GetNamespace("MAPI").GetDefaultFolder.olFolderInbox.Items
For Each itm In inbox
    If itm.Class = olMail Then
        Set mailitm = itm
        Debug.Print "E-Mail von " & mailitm.Sender
    ElseIf itm.Class = olMeetingRequest Then
        Set meetreq = itm
        Debug.Print "Einladung von " & meetreq.SenderName
    End If
    max = max - 1
    If max <= 0 Then
        Exit For
    End If
Next itm
End Sub
```

8.3.2 Nice to know: Unterschiedliche Objekttypen in einer Collection und type cast

Wenn ein Programm ein Objekt von einem Objekttyp in einen anderen Objekttyp überführt, bezeichnet man dies in der Fachsprache als „type cast“. Der „type cast“ ist für das Funktionieren eines VBA-Programms technisch nicht nötig, bringt aber Vorteile während der Entwicklung: Die Entwicklungsumgebung kann Methoden- und Feldaufrufe für das Objekt autovervollständigen, und typ-inkompatible Objektzuweisungen, Methodenaufrufe und Feldzugriffe fallen bei der ersten Programmausführung zuverlässig auf. Es ist daher gute Programmierpraxis, für Objekte und auch für elementare Daten immer Variablen des speziellstmöglichen passenden Objekttyps beziehungsweise Datentyps zu verwenden.

8.4 Demo-Anwendung „Posteingangsverwaltung“

Die Demo-Anwendung „Posteingangsverwaltung“ listet alle Elemente aus dem Outlook-Posteingang auf, die am aktuellen Tag dort eingegangen sind, und trägt den Betreff und den Eingangszeitpunkt in ein Excel-Arbeitsblatt ein (für Demozwecke auch die Art des Elements). Meetingeinladungen mit dem Betreff „Einladung: P4“ sagt sie automatisch ab und löscht die betreffenden Termine aus dem Kalender. Dazu ruft sie Methoden des Elementtyps `MeetingItem` auf.

Achtung: Beim Betrieb und bei der Entwicklung von Automatisierungen mit VBA ist Vorsicht geboten. Versendete E-Mail-Nachrichten oder Besprechungseinladungen, Zusagen oder Absagen von Meetings kommen bei den jeweils eingetragenen Empfängern an – Experimente mit Geschäftskontakten sollte man besser vermeiden.

Ein interessantes Detail der Demo-Anwendung ist der Filter, der die E-Mails des aktuellen Tags im Posteingang selektiert. Zum Filtern wird ein String mit Filterkriterien definiert. Der folgende Code-Schnipsel definiert den Filter-String und druckt ihn aus, um ihn überprüfen zu können.

```
Sub Jetfilter()
Dim filterkriterien As String
filterkriterien = "[ReceivedTime] > '"
& Format(Date, "DD-MM-YYYY hh:nn") & "'"
Debug.Print filterkriterien
End Sub
```

Der Filter-String setzt sich wie folgt zusammen:

- [ReceivedTime] ist das Feld, auf dem gefiltert wird
- Date liefert das jeweilige Tagesdatum
- Format ist eine Funktion, die einen Date-Wert als String ausgibt und nach einem angegebenen Muster formatiert. Das Muster ist hier "DD-MM-YYYY hh:nn". Darin bedeuten DD=Tag, MM=Monat, hh=Stunden, nn=Minuten in zweistelliger Darstellung, also gegebenenfalls mit führender Null. YYYY ergibt eine Jahreszahl mit vier Stellen.

Die Demo-Anwendung „Posteingangsverwaltung“ verwendet diesen Filter-String als Parameter für die Restrict-Methode der Collection Items:

```
CONST ABLEHN_BETREFF As STRING = "Einladung: P4"

Sub PosteingangBearbeiten()
Dim olApp As New Outlook.Application
Dim eingang As Outlook.MAPIFolder
Dim postheute As Items
Dim filterkriterien As String
Dim itm As Object
Dim meetreq As MeetingItem
Dim meeting As MeetingItem
Dim wsProt As Worksheet
Dim zeile As Integer: zeile = 1

Set wsProt = Worksheets.Add
On Error GoTo Fehler

filterkriterien = _
"[ReceivedTime] > '" & Format(Date, "DD-MM-YYYY hh:nn") & "'"
```

```
Set eingang = olApp.GetNamespace("MAPI")._
GetDefaultFolder(olFolderInbox)
Set postheute = eingang.Items.Restrict(filterkriterien)

For Each itm In postheute
    wsProt.Cells(zeile, 1).Value = Format(itm.ReceivedTime, "hh:nn")
    wsProt.Cells(zeile, 2).Value = itm.Subject
    wsProt.Cells(zeile, 3).Value = itm.Class
    If itm.Class = olMeetingRequest Then
        Set meetreq = itm
        If meetreq.Subject = ABLEHN_BETREFF Then
            Set meeting = meetreq.GetAssociatedAppointment(True) _
                .Respond(olMeetingDeclined, True)
            meeting.Send
            meetreq.GetAssociatedAppointment(True).Delete
            wsProt.Cells(zeile, 4) = "Abgelehnt"
        End If
    End If
    zeile = zeile + 1
Next itm
Exit Sub

Fehler:
Debug.Print Err.Number & ":" & Err.Description
Resume Next
End Sub
```

Der Einfachheit halber gibt die Demo-Anwendung etwaige Fehlermeldungen per `Debug.Print` aus. Sie könnte die Fehlermeldungen aber auch in eine Protokolldatei schreiben, sodass man Fehler erkennen und manuell nachbearbeiten kann.

8.5 Demo-Anwendung „Anhägeverwaltung“ mit komplexen Filtern

Die Demo-Anwendung „Anhägeverwaltung“ durchsucht den Posteingang nach E-Mails aus einem bestimmten Zeitraum und mit einem bestimmten Ausdruck im Betreff, die außerdem einen Anhang enthalten. Den jeweils ersten Anhang dieser E-Mails speichert sie in ein Dateiverzeichnis. Sie zeigt zwei interessante Themen: erstens die Erstellung komplexer Filter und zweitens die Berechnung von Datumsangaben. Diese beiden Themen werden vorab mit kleinen Code-Schnipseln demonstriert. In Abschn. 8.5.3 ist dann der komplette Code zu sehen.

8.5.1 Komplexe DASL-Filter

8.5.1.1 DASL-Filter und JET-Filter

Die `Restrict`-Methode der Collection `Items` akzeptiert zwei Arten von Filtern: JET-Filter und DASL-Filter. Die Demo-Anwendung „Posteingangsverwaltung“ in Abschn. 8.4 verwendet einen JET-Filter. DASL-Filter haben gegenüber JET-Filtern den Vorteil, dass sich darin sehr einfach Platzhalter (Wildcards) einsetzen lassen. Dafür sind sie etwas schwieriger zu erstellen. DASL-Filter haben eine SQL-ähnliche Syntax und beginnen immer mit der Zeichenfolge "@SQL=". Wie in SQL üblich werden Platzhalter durch Prozentzeichen % dargestellt statt durch die sonst üblichen Sternchen *.

In JET-Filtern gibt man die Element-Eigenschaften, auf denen gefiltert wird, durch ihre Bezeichner an, zum Beispiel `ReceivedTime`. DASL-Filter verwenden stattdessen die DASL-Codes der Element-Eigenschaften. Für die Demo-Anwendung „Anhängeverwaltung“ ergeben sich damit zum Beispiel folgende Filterkriterien:

- `filter1 = _
 """/schemas.microsoft.com/mapi/proptag/" & _
 "0x001a001e """='ipm.Note'"`
- `filter2 = _
 """/urn:schemas:httpmail:subject"" LIKE '%Protokoll%'"`
- `filter3 = _"""/urn:schemas:httpmail:datereceived"" > = " & _
 DreiMonateZurueck`
- `filter4 = _"""/urn:schemas:httpmail:hasattachment"" = 1"`

Sie haben folgende Bedeutung:

- `filter1`: Elemente mit Typ E-Mail.
- `filter2`: Betreff enthält die Zeichenkette „Protokoll“.
- `filter3`: Eingangsdatum liegt maximal 3 Monate zurück. Die Funktion, die das drei Monate zurückliegende Datum berechnet, wird in Abschn. 8.5.2 erläutert.
- `filter4`: E-Mail enthält Dateianhänge, 1 bedeutet hier „wahr“.

Die Herausforderung bei DASL-Filtern liegt darin, die DASL-Codes zu finden, denn sie sind sehr unvollständig dokumentiert. P. Hoff [4] hat jedoch ein Vorgehen gefunden, das dies sehr vereinfacht. Man nutzt dabei den Filter-Dialog von Outlook, den man über den Pfad **Ansicht > Ansichtseinstellungen > Filtern** erreicht, siehe Abb. 8.2. Im Reiter **Erweitert** des Filter-Dialogs definiert man Filterkriterien mit den Klartext-Bezeichnern der Element-Eigenschaften und wechselt dann auf den Reiter **SQL**. Dort findet man die DASL-Version der Filterkriterien und kann sie mit Copy&Paste in den VBA-Code übernehmen und, wenn nötig, noch bearbeiten.

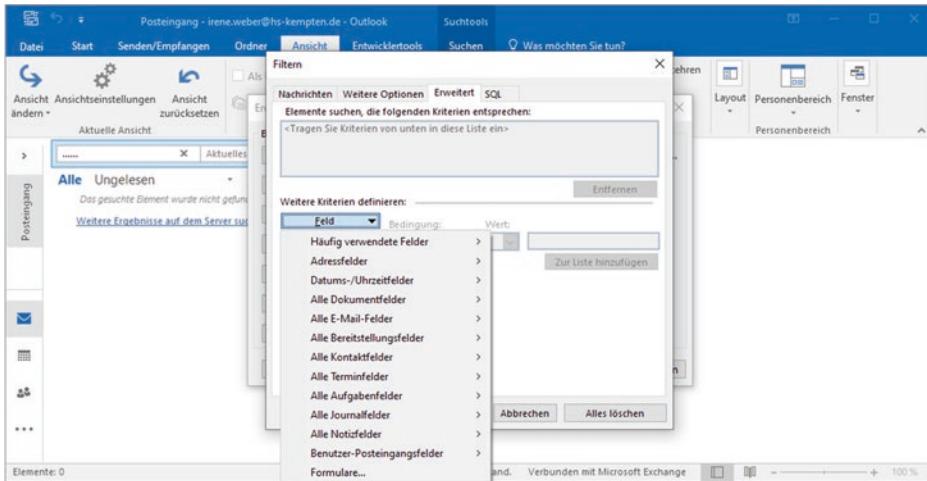


Abb. 8.2 Definition von DASL-Filtern

8.5.1.2 Komplexe Filter definieren

Sind komplexere Filter nötig, kann man Filterkriterien mit den Booleschen Operatoren And oder Or verknüpfen und mit Not negieren. Die `filter1` bis `filter4` der hier entwickelten Demo-Anwendung „Anhängeverwaltung“ sollen alle zugleich zutreffen und werden dazu mit And verknüpft. Der Befehl zur Definition des Filterstrings lautet damit:

- `filterKomp = "@SQL= " & filter1 & " And " & filter2 & _
" And " & filter3 & " And " & filter4`

Die Operatoren Not, And und Or funktionieren für DASL-Filter und JET-Filter. Die beiden Filterarten können aber nicht gemischt werden.

8.5.1.3 Nice to know: Anführungszeichen im String „escapen“

Filter definiert man im Stringformat. Der Filter-String enthält dabei weitere Strings, nämlich DASL-Codes und eventuell auch Suchbegriffe und Eigenschaftsnamen, die mit Anführungszeichen eingrenzt werden müssen. Es gibt also äußere Anführungszeichen, die den Filter-String im VBA-Code begrenzen, und darin eingeschlossen innere Anführungszeichen, die Bestandteile der Filterkriterien sind. Damit VBA die inneren Anführungszeichen von den äußeren unterscheiden kann, muss man sie entsprechend kennzeichnen. Man nennt dies in der Fachsprache „escapen“. In VBA markiert man die inneren Anführungszeichen durch ein weiteres Anführungszeichen, also "". In den Filterbeispielen `filter1` bis `filter4` ergeben sich damit Kombinationen von drei aufeinanderfolgenden Anführungszeichen.

8.5.2 Berechnung von Datumsangaben

Die Demo-Anwendung soll E-Mails bearbeiten, die maximal drei Monate alt sind. Das Datum des jeweils drei Monate zurückliegenden Tags wird mit der Funktion `Drei-MonateZurueck` berechnet. Die Funktion arbeitet mit `Date`. In der Sprache VBA bezeichnet das Schlüsselwort `Date` nicht nur einen Datentyp für Zeitangaben, es bezeichnet zugleich auch eine Funktion, die das aktuelle Tagesdatum als Wert vom Typ `Date` ausgibt. Eine verwandte Funktion ist `Now`, die den aktuellen Zeitpunkt ausgibt, ebenfalls als Wert vom Typ `Date`.

`DateAdd(Interval, Number, Date)` ist eine VBA-Funktion, die ausgehend von einem `Date`-Wert einen neuen `Date`-Wert berechnet. Neben dem Ausgangszeitpunkt sind dafür die Einheit des Intervalls und die Anzahl der Einheiten anzugeben, um die der neu berechnete Zeitpunkt vom Ausgangszeitpunkt abweichen soll. Mögliche Kürzel für Intervall-Einheiten sind in Tab. 8.3 zusammengestellt. Zum Beispiel rechnet der folgende Code-Schnipsel vom aktuellen Zeitpunkt aus eine halbe Stunde zurück und gibt den resultierenden Zeitpunkt formatiert aus. Kürzel des Format-Strings wurden bereits bei der Definition des JET-Filtern in Abschn. 8.4 verwendet. Sie ähneln zwar den Kürzeln für Intervall-Einheiten in Tab. 8.3, sind aber nicht identisch.

```
Sub HalbeStundeZurueck()
Dim d As Date

d = DateAdd("n", -30, Now)
Debug.Print Format(d, "DD-MM-YYYY hh:nn:ss")
End Sub
```

Um vom aktuellen Tag aus drei Monate zurückzurechnen, verwendet man den Befehl `DateAdd("m", -3, Date)`.

Tab. 8.3 Intervallkürzel für `DateAdd[5]`

Kürzel	Bedeutung
"d"	Day
"ww"	Week
"w"	Weekday
"m"	Month
"q"	Quarter
"yyyy"	Year
"y"	Day of the year
"h"	Hour
"n"	Minute
"s"	Second

8.5.3 Der Code der Demo-Anwendung

Die Demo-Anwendung „Anhägeverwaltung“ bearbeitet E-Mails, die mindestens einen Anhang besitzen. E-Mail-Anhänge haben im Outlook-Objektmodell den Objekttyp Attachment. Das Feld FileName eines Attachment-Objekts enthält den Namen der angehängten Datei. Die Methode SaveAsFile eines Attachment-Objekts speichert die Datei.

Die Demo-Anwendung speichert die Anhänge in das Verzeichnis, das auch die Excel-Arbeitsmappe mit der Demo-Anwendung enthält. Diese sollte deshalb vor der Ausführung des Codes bereits in ein geeignetes Verzeichnis gesichert worden sein.

```
Sub AnhaengeBearbeiten()
Dim protPfad As String: protPfad = ThisWorkbook.Path
Dim olApp As New Outlook.Application
Dim eingang As Outlook.Folder
Dim protokolle As Items

Dim filter1 As String, filter2 As String
Dim filter3 As String, filter4 As String
Dim filterKomplett As String
Dim mailitm As Object
On Error GoTo Fehler

filter1 = _
    """http://schemas.microsoft.com/mapi/proptag/0x001a001e"" " & _
    & "= 'ipm.Note'"
filter2 = _
    """urn:schemas:httpmail:subject"" LIKE '%Protokoll%'"
filter3 = _
    """urn:schemas:httpmail:datereceived"" >= " & DreiMonateZurueck
filter4 = """urn:schemas:httpmail:hasattachment"" = 1"
filterKomplett = "@SQL= " & filter1 & " And " & filter2 _
    & " And " & filter3 & " And " & filter4

Debug.Print filterKomplett

Set eingang = olApp.GetNamespace("MAPI"). _
    GetDefaultFolder(olFolderInbox)
Set protokolle = eingang.Items.Restrict(filterKomplett)

For Each mailitm In protokolle
    mailitm.Attachments(1).SaveAsFile protPfad & "\" _
```

```

    & mailitm.Attachments(1).Filename
Next mailitm

Set mailitm = Nothing
Exit Sub

Fehler:
If Err.Number <> 0 Then
    Debug.Print Err.Number & ":" & Err.Description
End If
Resume Next
End Sub

Function DreiMonateZurueck() As String
Dim d As Date

d = DateAdd("m", -3, Date)
DreiMonateZurueck = "" & Format(d, "DD-MM-YYYY hh:nn") & ""
End Function

```

Diese Demo-Anwendung wird flexibel, wenn man Betreff, den betrachteten Zeitraum, das Dateiverzeichnis zum Speichern der Anhänge und ähnliche Einstellungen auf einem Arbeitsblatt der Demo-Anwendung editierbar macht, ähnlich wie bei der Demo-Anwendung „PowerPoint-Generator“ im Abschn. 7.2. Aus Platzgründen ist dies hier nicht ausprogrammiert. Auch das Speichern von mehreren Dateianhängen einer E-Mail wäre noch zu ergänzen.

8.6 Demo-Anwendung „Termine-Controlling“

Die Demo-Anwendung „Termine-Controlling“ lädt Termine aus dem Outlook-Kalender in eine Excel-Tabelle, sodass man sie auswerten und sich einen Überblick über den Zeitaufwand für Meetings zu verschiedenen Themen verschaffen kann.

Diese Demo zeigt mit GetTable eine Alternative zum Filtern mit Restrict wie in den vorhergehenden Demos. GetTable ist eine Methode des Objekttyps Folder und akzeptiert wie Restrict als Parameter einen String mit Filterkriterien. Das Ergebnis ist ein Table-Objekt. Der Code zeigt, wie sich die Tabelle auf interessierende Spalten reduzieren und durchlaufen lässt.

```

Sub FindAppts()
Dim olApp As New Outlook.Application

```

```
Dim kalender As Outlook.Folder
Dim treffenTabelle As Outlook.Table
Dim besprechungen As Outlook.Items
Dim treffen As Outlook.Row

Dim filter1 As String, filter2 As String, filterKomp As String
Dim zeile As Integer: zeile = 1

Set kalender = olApp.Session.GetDefaultFolder(olFolderCalendar)

filter1 = _
    """urn:schemas:calendar:dtstart"" >= '2023/01/01'"
filter2 = _
    """urn:schemas:calendar:dtstart"" <= '2023/12/31'"
filterKomp = "@SQL= " & filter1 & " And " & filter2
Set treffenTabelle = kalender.GetTable(filterKomp)

With treffenTabelle.Columns
    .RemoveAll
    .Add ("Subject")
    .Add ("Start")
    .Add ("Duration")
End With

Dim wks As Worksheet
Set wks = Worksheets.Add

wks.Cells(zeile, 1) = "Subject"
wks.Cells(zeile, 2) = "Start"
wks.Cells(zeile, 3) = "Duration"

Do Until (treffenTabelle.EndOfTable)
    zeile = zeile + 1
    Set treffen = treffenTabelle.GetNextRow()
    wks.Cells(zeile, 1) = treffen("Subject")
    wks.Cells(zeile, 2) = treffen("Start")
    wks.Cells(zeile, 3) = treffen("Duration")
Loop
End Sub
```

Diese und die vorhergehenden Demo-Anwendungen arbeiten mit Elementen, die in Outlook-Ordnern bereits vorhanden sind. Die folgende Demo-Anwendung erzeugt neue Elemente, nämlich E-Mails.

8.7 Demo-Anwendung „Mailing-Tool“

Die Demo-Anwendung „Mailing-Tool“ versendet formatierte und individualisierte E-Mails an eine Liste von Empfängern. Sie läuft in Excel als Wirtsanwendung und verwendet neben Outlook auch die Objektbibliothek von Word, weil sie die Nachrichten mithilfe eines Word-Dokuments erstellt. Zunächst demonstriert ein Skript die grundlegenden Befehle, um eine neue E-Mail zu erzeugen und zu versenden. Es wird dann erweitert zur Demo-Anwendung „Mailing-Tool“.

8.7.1 Eine einfache E-Mail erstellen und versenden

Das folgende Skript sendet eine E-Mail an einen einzelnen Empfänger. Es erzeugt ein neues Outlook-MailItem und befüllt es mit einigen Daten. Dann lässt es Outlook die E-Mail anzeigen, sodass sie vor dem Absenden geprüft werden kann. Das Demo-Skript enthält eine Sperre, die den Vorgang abbricht, wenn Outlook nicht aktiv ist. Zwar kann ein Skript mithilfe des Outlook-Objektmodells auch dann ein MailItem erstellen und anzeigen, wenn Outlook nicht aktiv läuft. Es kann das MailItem dann allerdings nicht versenden, sondern lediglich im Outlook-Postausgang speichern. Dort gespeicherte E-Mails versendet Outlook beim nächsten Start automatisch. Falls dieses Verhalten gewünscht ist, kann man die Sperre entfernen.

```
Sub EinfacheMailVersenden()
Dim outlookApp As New Outlook.Application
Dim mailItm As Outlook.MailItem

' Abbruch, wenn Outlook nicht läuft
On Error GoTo Outlook_Fehler
' Test, ob Outlook läuft:
Set outlookApp = GetObject(, "Outlook.Application")
' Fehlerabfangen wieder ausschalten
On Error GoTo 0

Set mailItm = outlookApp.CreateItem(0)
With mailItm

    .To = "jemand@sonstwo.de"
    .Subject = "Test"
    .Body = "Hallo, dies ist ein Test."
End With
```

```
mailItm.Display
If (vbYes = MsgBox("Mails versenden?", vbYesNo)) Then
    mailItm.Send
Else
    mailItm.Close (oldDiscard) ' Schließen ohne Speichern
End If
Exit Sub

Outlook_Fehler:
MsgBox ("Abbruch - Bitte Outlook starten und Skript neu ausführen.")
End Sub
```

Um E-Mails für mehrere Adressaten mit jeweils individuellen Inhalten zu erzeugen, bettet man den Demo-Code in eine Schleife ein wie im nächsten Abschnitt.

8.7.2 Formatierte E-Mails versenden

Die Demo-Anwendung „Mailing-Tool“ sendet formatierte E-Mails an Empfänger, die auf einem Excel-Arbeitsblatt hinterlegt sind. Als Vorlage dient ein formatiertes Word-Dokument mit Textmarken wie in Abb. 8.3. Das Demo-Szenario besteht darin, Teilnahmebestätigungen an Kursteilnehmer zu versenden. Die Textmarken heißen daher „teilnehmername“ und „kursname“. Das Skript ersetzt die Textmarken für jede E-Mail durch individuelle Werte, die es zusammen mit den E-Mail-Adressen der Empfänger von einem Arbeitsblatt des Mailing-Tools liest. Das Arbeitsblatt sollte den Namen „Adres-

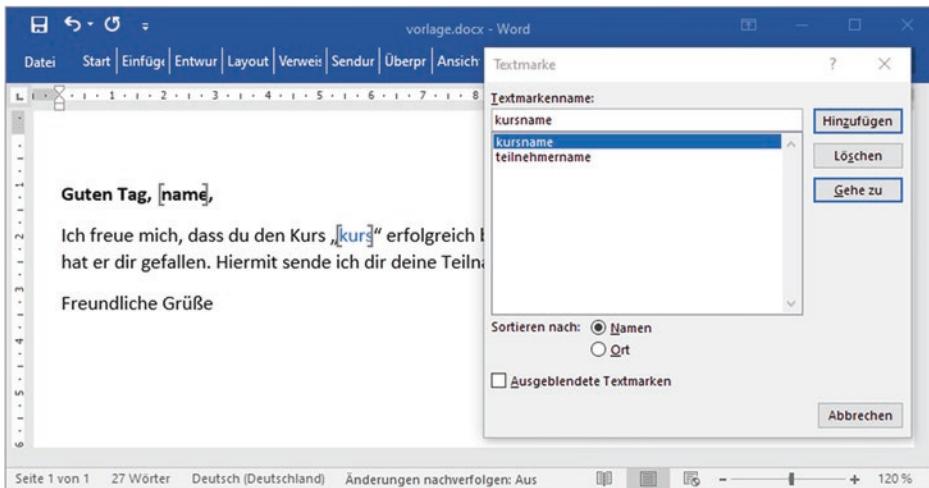


Abb. 8.3 Word-Dokument als Vorlage für die Demo-Anwendung „Mailing-Tool“

	A	B	C	D	E	F
1	Email	Betreff	Name	Kurs		
2	anna@irgendwo.de	Teilnahmebescheinigung	Anna Beta	Business English		
3	cyril@c-delta.de	Zertifikat	Cyril Delta	Projektmanagement		
4	cyril@c-delta.de	Zertifikat	Cyril Delta	Projektmanagement 2	Senden	
5	eva@nirgends.eu	Teilnahmebescheinigung	Eva Gamma	Chinesisch für Anfänger		
6	eva@nirgends.eu	Zertifikat	Eva Gamma	Projektmanagement		
7						

Abb. 8.4 Excel-Arbeitsblatt der Demo-Anwendung „Mailing-Tool“

saten“ haben. Ein Beispiel-Arbeitsblatt ist in Abb. 8.4 zu sehen. Jede Zeile enthält eine Empfänger-Adresse, einen Betreff, einen Teilnehmernamen und eine Kursbezeichnung. Die erste Zeile zeigt Überschriften. Beginnend mit der zweiten Zeile erzeugt das Mailing-Tool für jede Zeile eine E-Mail, bis es eine Zeile ohne Daten findet.

Das Demo-Skript bietet die Chance, die erste E-Mail vor dem Versenden zu prüfen und den E-Mail-Versand gegebenenfalls abzubrechen.

```

Const VORLAGE as String = "vorlage.docx"

Sub MailFormatiert()
Dim outlookApp As New Outlook.Application
Dim wordApp As New Word.Application
Dim wordDoc As Word.Document
Dim mailDoc As Word.Document
Dim mailItm As Outlook.MailItem
Dim addressrow As Integer

Dim istBestaetigt As Boolean

On Error GoTo Aufraeumen
Set wordDoc = wordApp.Documents. _
    Open(ThisWorkbook.Path & "\" & VORLAGE)
wordDoc.Content.Copy

addressrow = 2
With Worksheets("Adressaten")
    Do While .Cells(addressrow, 1) <> ""
        Set mailItm = outlookApp.CreateItem(0)
        mailItm.BodyFormat = olFormatRichText

```

```
mailItm.To = .Cells(addressrow, 1).Value
mailItm.Subject = .Cells(addressrow, 2).Value
Set mailDoc = mailItm.GetInspector.WordEditor
mailDoc.Content.Paste      ' MailItem enthält jetzt die Vorlage
mailDoc.Bookmarks("teilnehmernname").Range.Text = _
.Cells(addressrow, 3)
mailDoc.Bookmarks("kursname").Range.Text = .Cells(addressrow, 4)

If Not istBestaetigt Then
    mailItm.Display
    If (vbYes = MsgBox("Mails versenden?", vbYesNo)) Then
        istBestaetigt = True
    Else
        mailItm.Close (oldDiscard)   ' ohne Speichern schließen
        Exit Do
    End If
End If
mailItm.Display
mailItm.Send
addressrow = addressrow + 1
Loop
End With

Aufräumen:
If Err.Number <> 0 Then
    MsgBox (Err.Number & Chr(13) & Err.Description)
End If

On Error Resume Next      ' Word auf alle Fälle wieder schließen
wordDoc.Close (False)
wordApp.Quit
Set wordApp = Nothing
End Sub
```

Dieser Code hat zwei Auffälligkeiten: Erstens überträgt er den Inhalt der Word-Vorlage mit Copy&Paste in den WordEditor des MailItem-Objekts. Zweitens zeigt er jedes MailItem mit seiner Display-Methode an, bevor er es abschickt. Beides wirkt vielleicht umständlich programmiert, ist aber so erforderlich.

Das Mailing-Tool realisiert einen E-Mail-Serienbrief mit VBA. Zwar bringt Word bereits eine eingebaute Serienbrief-Funktion mit. Ein VBA-basiertes Mailing-Tool hat demgegenüber die Vorteile, dass es sich leicht mit anderen VBA-Tools integrieren oder für speziellere Aufgaben anpassen lässt.

8.8 Fazit und Ausblick

Die Integration von Outlook mit anderen Anwendungen eröffnet vielfältige Möglichkeiten, etwa beim Verwalten oder Erstellen von E-Mails, Terminen, Kontakten und Aufgaben. Informationen und Dateien, die regelmäßig in Outlook ankommen, können effizient in andere Anwendungen übernommen und bearbeitet werden. Auch spezielle Tools für kleine, sich wiederholende Routineaufgaben sind denkbar, zum Beispiel aus einem in Excel erstellten Projektplan automatisch Outlook-Termine generieren, ein in Word erstelltes Meeting-Protokoll automatisiert an darin erfasste Teilnehmer versenden, aus der Titelfolie einer PowerPoint-Präsentation eine Einladungs-E-Mail generieren und vieles mehr.

Dieses Kapitel hat sich damit befasst, Outlook aus Excel heraus anzusteuern. Im folgenden Kapitel werden VBA-Automatisierungen direkt in Outlook entwickelt und in seine Bedienoberfläche eingebunden.

Literatur

1. o365devx, „NameSpace.Logon-Methode (Outlook)“. <https://docs.microsoft.com/de-de/office/vba/api/outlook.namespace.logon> (zugegriffen 8. November 2021).
2. o365devx, „OlDefaultFolders enumeration (Outlook)“. <https://docs.microsoft.com/en-us/office/vba/api/outlook.oldefaultfolders> (zugegriffen 9. November 2021).
3. o365devx, „OlItemType enumeration (Outlook)“. <https://docs.microsoft.com/en-us/office/vba/api/outlook.olitemtype> (zugegriffen 19. Oktober 2021).
4. P. Hoff, „Finding DASL Property Names“, *Anything Else I Can Think Of*, 19. Dezember 2008. <http://philliphoff.github.io/finding-dasl-property-names/> (zugegriffen 25. Oktober 2021).
5. o365devx u. a., „DateAdd-Funktion (Visual Basic für Anwendungen)“, *Microsoft Learn Office VBA*, 19. Februar 2023. <https://learn.microsoft.com/de-de/office/vba/language/reference/user-interface-help/dateadd-function> (zugegriffen 25. März 2023).



Outlook II und MS Forms – Interaktive Tätigkeiten unterstützen

9

Inhaltsverzeichnis

9.1	VBA-Entwicklung in Outlook	154
9.2	Elemente des Outlook-Objektmodells II	154
9.2.1	Die Objekte Explorer und Inspector	155
9.2.2	Die Objekte ActiveWindow, ActiveExplorer und ActiveInspector.....	155
9.3	Demo-Anwendung „Dateianhänge-Auswahl v1“	156
9.3.1	Auswahlformular als UserForm	157
9.3.2	VBA-Code im Modul AnhaengeAuswaehlen	159
9.3.3	Integration von Makros in die Outlook-Bedienoberfläche	160
9.4	Elemente des Outlook-Objektmodells III – Der Outlook-Speicher.....	162
9.4.1	Die Objekttypen StorageItem und UserProperty.....	163
9.4.2	Code-Beispiel zum Outlook-Speicher.....	163
9.5	Demo-Anwendung „Dateianhänge-Auswahl v2“	165
9.5.1	Konfiguration des Verzeichnispfads per UserForm	165
9.5.2	VBA-Code des Moduls AnhangKonfig.....	166
9.5.3	Anpassungen der UserForm AnhaengeAuswahlForm	167
9.5.4	Modul AnhaengeAuswaehlen.....	168
9.6	Demo-Anwendung „Dateianhänge-Auswahl v3“	170
9.6.1	Aufbau der Demo-Anwendung „Dateianhänge-Auswahl v3“.....	171
9.6.2	UserForm AnhaengeDynForm in der Entwurfsphase	172
9.6.3	Code der UserForm AnhaengeDynForm.....	172
9.6.3.1	Sub UserForm_Initialize der UserForm AnhaengeDynForm ...	173
9.6.3.2	Code des „Ok“-Button mit Verwendung einer Collection	175
9.6.3.3	Code des „Konfig“-Button	176
9.6.4	Modul AnhangKonfigPicker.....	176
9.6.5	Modul AnhaengeAuswaehlen v3.....	178
9.7	Fazit	179
	Literatur.....	179

Automatisierungen und Erweiterungen von Anwendungssystemen sind besonders hilfreich und benutzerfreundlich, wenn sie sich reibungslos in die gewohnten Arbeitsabläufe einfügen. In diesem Kapitel wird eine Erweiterung für Outlook entwickelt und in das Outlook-Menüband integriert. Sie soll Mitarbeiter unterstützen, die häufig E-Mails mit bestimmten Dateianhängen erstellen, indem sie die Dateien in einem leicht erreichbaren Formular anzeigen. Um eine Datei an die E-Mail anzuhängen, hakt man sie einfach im Formular an.

Das Kapitel entwickelt drei Versionen der Anwendung. Jede Version ist für sich lauffähig und demonstriert interessante VBA-Techniken. Die erste Version verwendet Outlook-Objekte, die zur Bedienoberfläche von Outlook gehören, und erweitert die Bedienoberfläche von Outlook um eine UserForm. In der zweiten Version kommt der interne Speicher von Outlook zum Einsatz, nämlich der Objekttyp `StorageItem`. Die dritte Version nutzt die Objektbibliothek MS Forms 2.0, die das Objektmodell für UserForms und Steuerelemente enthält, um dynamisch eine UserForm zu erzeugen. Das bedeutet, dass das VBA-Skript die UserForm erst dann komplett erstellt, wenn der Benutzer sie öffnet. Dabei passt es die UserForm an die aktuelle Situation an.

9.1 VBA-Entwicklung in Outlook

Bei der VBA-Entwicklung in Outlook gibt es eine Besonderheit: Outlook greift immer nur auf ein bestimmtes VBA-Projekt zu. Dieses trägt die Dateibezeichnung „`VbaProject.OTM`“ und liegt an einem vorgegebenen Pfad im Dateisystem, je nach Windows-Version zum Beispiel unter `C:\Users\<User>\AppData\Roaming\Microsoft\Outlook`. (Statt `<User>` ist der jeweilige Windows-User einzusetzen.) Wenn mehrere User auf einem PC mit Outlook arbeiten, findet so jeder automatisch seine persönlichen VBA-Makros und Anpassungen.

Genau wie Excel, Word und andere Office-Anwendungen öffnet auch Outlook die bekannte integrierte Entwicklungsumgebung für VBA. In Outlook zeigt die Entwicklungsumgebung jedoch immer nur das Projekt „`VbaProject.OTM`“. Es gibt keine Dialoge, um andere VBA-Projekte zum Öffnen auszuwählen oder das vorhandene VBA-Projekt unter einem anderen Namen abzuspeichern. Lediglich Export- und Importfunktionen für Formulare und Code-Module sind vorhanden. Man kann dennoch unterschiedliche Versionen des Outlook-VBA-Projekts abspeichern, zwischen verschiedenen VBA-Projekten wechseln oder ein VBA-Projekt an andere PCs oder User weitergeben, indem man die Datei „`VbaProject.OTM`“ umbenennt oder kopiert. [1]

9.2 Elemente des Outlook-Objektmodells II

Wichtige Objekttypen des Outlook-Objektmodells für dieses Kapitel sind `Explorer` und `Inspector`. Sie zeigen die Outlook-Elemente `MailItem`, `MeetingItem` etc. an.

9.2.1 Die Objekte Explorer und Inspector

Das Explorer-Objekt ist das Hauptfenster der Anwendung Outlook. Dort findet ein angemeldeter Benutzer seine Ordner und Elemente wie Posteingang oder Kalender mit E-Mails, Einladungen, Terminen und so weiter. Im Explorer kann man ein oder mehrere Elemente anwählen. Ein einzelnes Element kann man öffnen und direkt im Explorer bearbeiten.

Alternativ lassen sich Elemente auch in losgelösten Anzeigefenstern öffnen. Solche Anzeigefenster heißen im Outlook-Objektmodell Inspector. Für die verschiedenen Elementtypen gibt es entsprechende Inspector-Typen. Ein Inspector bietet in seinem Feld CurrentItem Zugriff auf das Element, das er gerade anzeigt.

Das Outlook-Objekt Explorer besitzt ein Feld Selection. Darin verwaltet es die Elemente, die der Benutzer im gerade offenen Ordner selektiert hat, wobei es das erste der selektierten Elemente im Explorer-Fenster anzeigt.

9.2.2 Die Objekte ActiveWindow, ActiveExplorer und ActiveInspector

Das Application-Objekt von Outlook besitzt die Felder ActiveWindow, ActiveExplorer und ActiveInspector. Sie verweisen auf die Komponenten, die aktuell aktiv sind. Das ActiveWindow kann dabei je nach Situation auf ein Explorer-Objekt oder auf ein Inspector-Objekt verweisen.

Der folgende Code-Schnipsel zeigt, wie ein Skript auf das Item zugreifen kann, das der Benutzer im aktiven Fenster offen hat. Dabei kann es sich um den Explorer oder um einen Inspector handeln.

```
Sub OffenesElementZugreifen()
Dim elem As Object

If TypeName(ActiveWindow) = "Explorer" Then
    If Not ActiveExplorer.Selection.Count = 0 Then
        Set elem = ActiveExplorer.Selection.Item(1)
    Else
        Debug.Print "Aktuell ist kein Element aktiv"
        Exit Sub
    End If
ElseIf TypeName(ActiveWindow) = "Inspector" Then
    Set elem = ActiveInspector.CurrentItem
End If
```

```

Debug.Print "Aktuelles Element: ' " & elem.Subject _
& " (Typ: " & elem.Class & ")"
End Sub

```

Die Variable `elem` ist hier mit dem Typ `Object` deklariert, damit sie jeden möglichen Objekttyp (E-Mail, Kalendereintrag etc.) aufnehmen kann.

9.3 Demo-Anwendung „Dateianhänge-Auswahl v1“

Die Demo-Anwendung „Dateianhänge-Auswahl“ unterstützt Benutzer dabei, Anhänge für E-Mails zusammenzustellen. Die Dateien stammen dabei aus einem festen Dateien-Pool. Beim Verfassen einer E-Mail öffnen Benutzer ein Formular und haken die benötigten Dateien darin an.

Die Demo-Anwendung besteht aus drei Komponenten: einer UserForm `AnhaengeAuswahlForm`, einem Code-Modul `AnhaengeAuswaehlen` und Erweiterungen am Outlook-Menüband. In Abb. 9.1 sind die VBA-Komponenten der Demo-Anwendung in der Entwicklungsumgebung zu sehen: Der **Projekt-Explorer** rechts zeigt die UserForm `AnhaengeAuswahlForm` und das Code-Modul `AnhaengeAuswaehlen`. Damit die Demo-Anwendung funktioniert, muss außerdem ein Dateiverzeichnis mit drei passenden Dokumenten als Dateianhänge vorhanden sein.

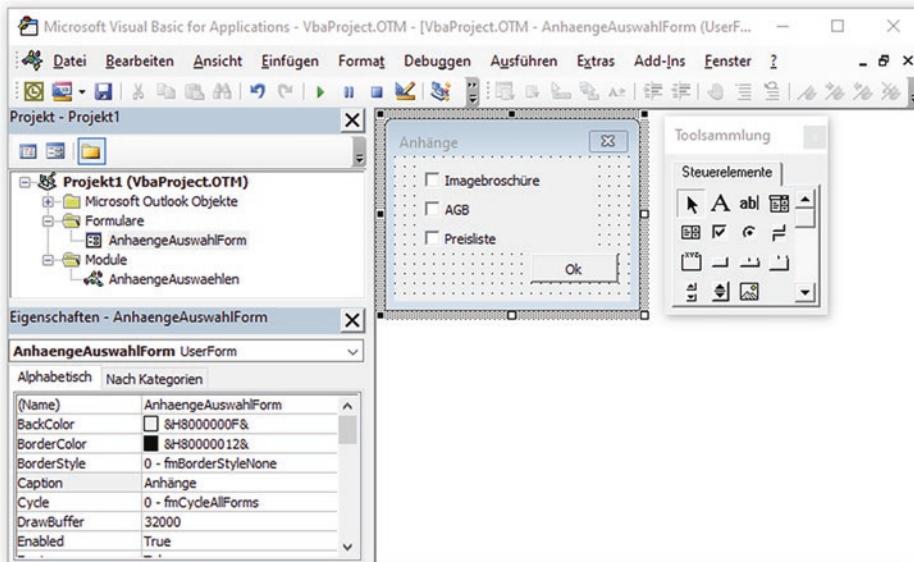


Abb. 9.1 UserForm `AnhaengeAuswahlForm` für die Auswahl von Dateianhängen in Outlook

9.3.1 Auswahlformular als UserForm

Outlook bietet einen Formulareditor, mit dem man eigene Formulare für die verschiedenen Elementtypen gestalten und mit Anzeige- und Eingabemöglichkeiten, Wertevorbelegungen und Steuerelementen ausstatten kann. Er ist auf der Registerkarte **Entwickertools** von Outlook zu finden. Da damit erstellte Formulare jedoch wenig mit VBA integriert sind, setzt diese Demo-Anwendung eine gewöhnliche VBA-UserForm ein, wie sie auch in anderen Office-Anwendungen zur Verfügung steht.

Eine neue UserForm erstellt man im **Projekt-Explorer** analog zum Einfügen eines neuen Moduls, also durch Rechtsklick in das **Projekt-Explorer**-Fenster, dann **Einfügen > UserForm**. Dabei öffnet sich zugleich auch die Toolsammlung mit den Steuerelementen, mit denen man eine UserForm gestaltet. Der Screenshot in Abb. 9.1 zeigt die fertige UserForm für die Dateianhänge-Auswahl in der Entwicklungsumgebung. Die Toolsammlung mit den verfügbaren Steuerelementen ist ebenfalls zu sehen. Wichtig ist der Name der UserForm, hier **AnhaengeAuswahlForm**, der im Feld **(Name)** des **Eigenschaften**-Fensters festgelegt wird. Mit diesem Namen spricht der VBA-Code die UserForm an. Die Beschriftung „Anhänge“ der UserForm lässt sich im Feld **Caption** des **Eigenschaften**-Fensters angeben, siehe Abb. 9.1.

Die UserForm für die Dateianhänge-Auswahl enthält drei CheckBoxes und eine Schaltfläche (Button). Sie werden per Drag&Drop aus der Toolsammlung auf der UserForm platziert. Ihre Details spezifiziert man ebenfalls im **Eigenschaften**-Fenster, siehe Abb. 9.2. Wichtig sind wieder der Name sowie die Beschriftung im Feld **Caption**. Hier sind die Namen der CheckBoxes einfach als **CheckBox1** bis **CheckBox3** belassen. Als Beschriftungen trägt man in die **Caption**-Felder sinnvollerweise die Dateibezeichnungen der Anhang-Dateien ein. Die Schaltfläche heißt **btnFertig**, siehe Abb. 9.2 rechts.

Im Codefenster der UserForm wird der VBA-Code definiert, der bestimmt, wie sich die UserForm und ihre Steuerelemente verhalten. Zum Öffnen des Codefensters klickt man mit der rechten Maustaste auf den Namen der UserForm im **Projekt-Explorer** und wählt im aufklappenden Menü den Eintrag **Code anzeigen**. Das Codefenster der UserForm ist in Abb. 9.3 zu sehen.

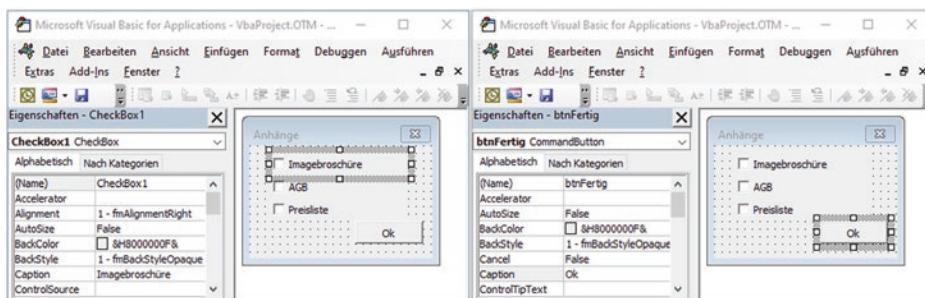


Abb. 9.2 Elemente der AnhaengeAuswahl-UserForm in Outlook

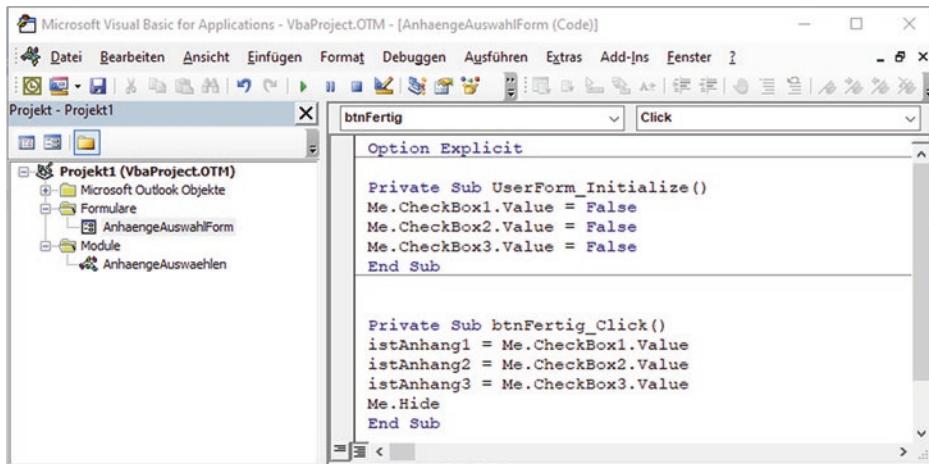


Abb. 9.3 Code der AnhaengeAuswahlForm-UserForm in Outlook

Die UserForm AnhaengeAuswahlForm benötigt nur zwei einfache Prozeduren. Die Prozedur UserForm_Initialize läuft bei jedem Öffnen der UserForm. Sie setzt den Wert aller CheckBox-Steuerelemente auf False.

```

Private Sub UserForm_Initialize()
    Me.CheckBox1.Value = False
    Me.CheckBox2.Value = False
    Me.CheckBox3.Value = False
End Sub

```

Die Prozedur btnFertig_Click startet, wenn der Benutzer die Schaltfläche btnFertig klickt. Wie unten zu sehen ist, überträgt sie den aktuellen Wert jeder Checkbox in eine Variable. Die Variablen istAnhang1 bis istAnhang3 sind als Public-Variablen im Modul AnhaengeAuswaehlen deklariert. Sie dienen der Kommunikation zwischen diesem Modul und der UserForm. Der Befehl Me.Hide schließt die UserForm.

```

Private Sub btnFertig_Click()
    istAnhang1 = Me.CheckBox1.Value
    istAnhang2 = Me.CheckBox2.Value
    istAnhang3 = Me.CheckBox3.Value
    Me.Hide
End Sub

```

9.3.2 VBA-Code im Modul AnhaengeAuswaehlen

Weiterer Code des Anwendungsbeispiels wird in einem Modul AnhaengeAuswaehlen erstellt. Das Modul beginnt mit einigen Deklarationen. Die Namen der Anhang-Dokumente und ihr Verzeichnispfad sind als Konstanten im Modul fest hinterlegt. Außerdem sind hier drei Public-Variablen deklariert, die in der UserForm ebenfalls sichtbar sind. Sie dienen zum Informationsaustausch zwischen der UserForm und den anderen Modulen.

```
Const verzeichnispfad As String = "C:\Users\...\...\Anhaenge\"  
Const anhang1 As String = "Imagebroschüre.pdf"  
Const anhang2 As String = "AGB.pdf"  
Const anhang3 As String = "Preisliste.pdf"  
  
Public istAnhang1 As Boolean  
Public istAnhang2 As Boolean  
Public istAnhang3 As Boolean
```

Das Modul AnhaengeAuswaehlen enthält die Prozedur AnhangAuswaehlen, die Anhänge an eine Nachricht anhängt. Zuerst muss die Prozedur auf diese Nachricht zugreifen. Dazu prüft sie, ob sie aus dem Outlook-Explorer oder aus einem losgelösten Nachrichten-Inspector gestartet wurde, also ob das aktuelle ActiveWindow der Explorer oder ein Inspector ist. Entsprechend entnimmt sie die Nachricht dem Explorer oder dem Inspector. Dann startet die Prozedur die AnhaengeAuswahl-UserForm. Jetzt kann der Benutzer darin Häkchen setzen und mit dem Button **Ok** die Auswahl abschließen. Die Prozedur btnFertig_Click der UserForm überträgt daraufhin die Auswahl in die Variablen istAnhang1 bis istAnhang3. Diese Variablen steuern jetzt, ob ein Dokument an die Nachricht angehängt wird.

```
Sub AnhangAuswaehlen()  
Dim auswahlForm As AnhaengeAuswahlForm  
Dim nachricht As MailItem  
' auf die Nachricht zugreifen  
If TypeName(ActiveWindow) = "Explorer" Then  
    If Not ActiveExplorer.ActiveInlineResponse Is Nothing Then  
        Set nachricht = ActiveExplorer.ActiveInlineResponse  
    End If  
ElseIf TypeName(ActiveWindow) = "Inspector" Then  
    If TypeOf ActiveInspector.CurrentItem Is Outlook.MailItem Then  
        Set nachricht = ActiveInspector.CurrentItem  
    End If  
End If
```

```

If nachricht Is Nothing Then      ' sollte nie vorkommen
    MsgBox ("Funktion Anhänge ist hier nicht möglich.")
    Exit Sub
End If

' Auswahlform anzeigen
Set auswahlForm = New AnhaengeAuswahlForm
auswahlForm.Show
' Benutzer bedient die UserForm ...
' ... jetzt sind in der UserForm die Variablen istAnhang_* gesetzt
If istAnhang1 Then
    nachricht.Attachments.Add verzeichnispfad & anhang1
End If
If istAnhang2 Then
    nachricht.Attachments.Add verzeichnispfad & anhang2
End If
If istAnhang3 Then
    nachricht.Attachments.Add verzeichnispfad & anhang3
End If
Set nachricht = Nothing
Set auswahlForm = Nothing
End Sub

```

Damit ist die Grundfunktion der Demo-Anwendung realisiert. Sie wird jetzt in das Menüband von Outlook integriert.

9.3.3 Integration von Makros in die Outlook-Bedienoberfläche

Um das Makro AnhangAuswaehlen aufrufen zu können, wird es in das Menüband eingebunden. Dies ist an zwei Stellen sinnvoll: in der Werkzeug-Registerkarte **Verfassen-tools** des Outlook-Explorers und in der Hauptregisterkarte **Neue Nachricht** eines Nachrichten-Inspectors. Abb. 9.4 zeigt, wie die Schaltfläche **Anhänge** zum Aufruf des Makros im Menüband der Registerkarte **Nachricht** des Nachrichten-Inspectors erscheint.

Um das Makro einzufügen, klickt man mit der rechten Maustaste in das Menüband, das das Makro aufnehmen soll. Dies öffnet eine Auswahlbox, die unter anderem den Befehl **Menüband anpassen** enthält. Dieser Befehl führt in den Dialog **Outlook-Optionen**.

Abb. 9.5 zeigt den Dialog **Outlook-Optionen** für das Menüband eines Nachrichten-Inspectors. Dort wird in der Registerkarte **Neue E-Mail-Nachricht** eine neue Gruppe angelegt und in **Extras** umbenannt. In diese Gruppe wird dann das Makro eingefügt. Wie Abb. 9.5 zeigt, findet man das Makro im Auswahlbereich unter **Befehle auswählen**, wenn dort die Option **Makros** eingestellt ist. Die Abbildung zeigt auch, wie man im Dia-

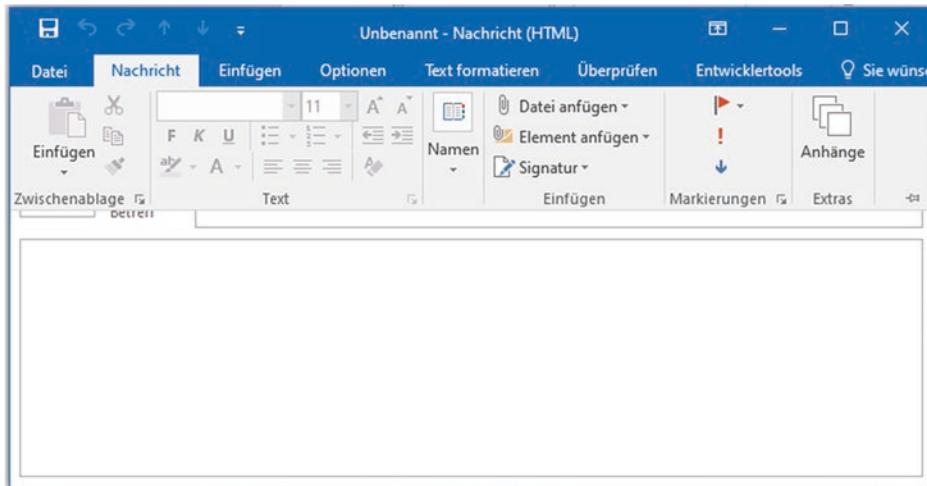


Abb. 9.4 Aufruf des Makros AnhangAuswaehlen im Menüband des **Nachricht**-Dialogs

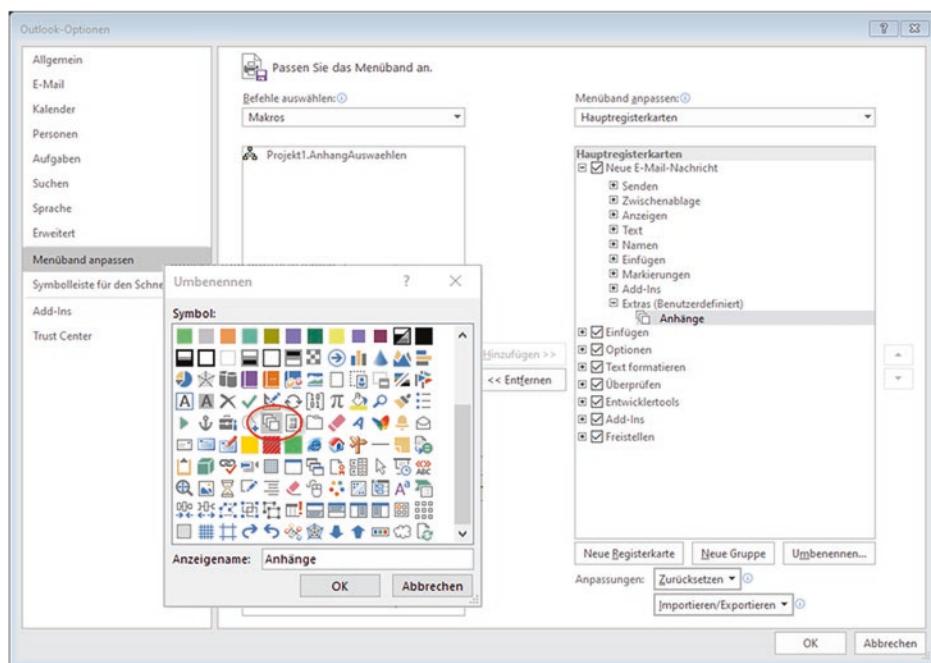


Abb. 9.5 Einbinden eines Makros in ein Menüband (Hauptregisterkarte **Neue E-Mail-Nachricht** des Nachrichten-**Inspector**)

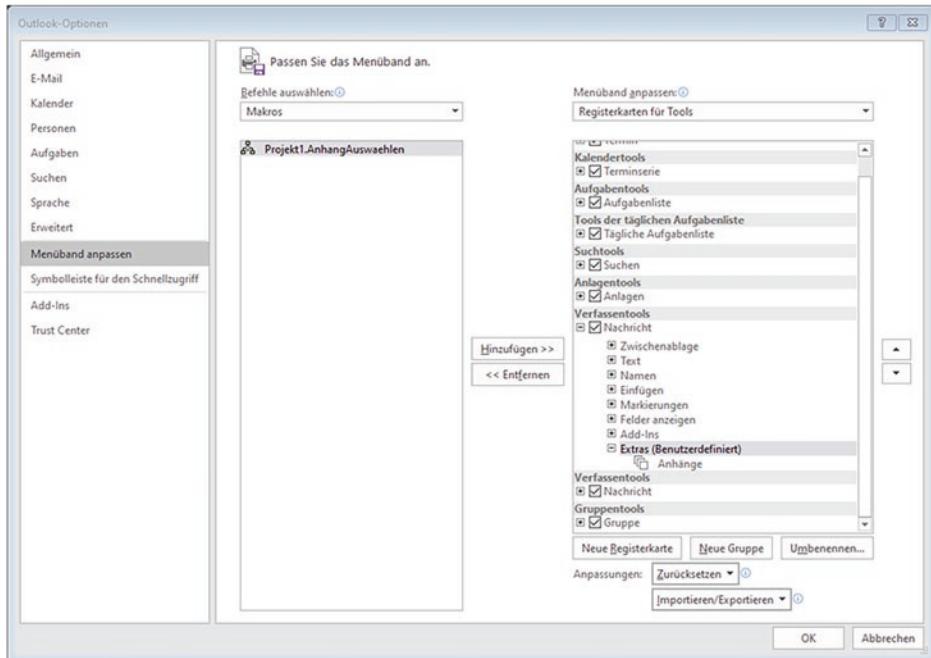


Abb. 9.6 Einbinden eines Makros in die Registerkarte **Verfassentools** des Outlook-Explorer

log **Umbenennen** einen Anzeigenamen und ein Symbol angeben kann, mit denen sich das eingebundene Makro im Menüband zeigt, vergleiche Abb. 9.4.

Das Menüband des **Explorer**-Fensters wird genauso angepasst, mit einem Unterschied: Nachdem ein Klick ins Menüband des **Explorer**-Fensters den Dialog **Outlook-Optionen** geöffnet hat, wird die Anzeige von den **Hauptregisterkarten** auf die **Registerkarten für Tools** umgestellt. So lässt sich nun der Makro-Aufruf in die Registerkarte **Verfassentools** einfügen, wie in Abb. 9.6 zu sehen ist.

9.4 Elemente des Outlook-Objektmodells III – Der Outlook-Speicher

In der bisher entwickelten Version der Demo-Anwendung „Dateianhänge-Auswahl v1“ sind der Dateipfad der anzuhängenden Dokumente und ihre Dateinamen im VBA-Code fest einprogrammiert. Um sie leicht finden und ändern zu können, stehen sie im Modul als Konstanten ganz oben. Für User, die den VBA-Code selbst editieren können, ist dies eine gangbare Lösung. Für weniger programmierfreudige Benutzer ist es besser, wenn sie Einstellungen über die Bedienoberfläche der Anwendung anpassen können statt im Code. Dafür ist es hilfreich, dass Outlook einen Speicher für solche Benutzerein-

stellungen bereithält. Die folgende Version der Demo-Anwendung zeigt, wie eine Einstellung für eine Outlook-Lösung im Outlook-Speicher dauerhaft hinterlegt und in einer UserForm zugänglich und editierbar gemacht wird.

9.4.1 Die Objekttypen StorageItem und UserProperty

Das Outlook-Objektmodell enthält mit dem `StorageItem` einen weiteren Elementtyp neben den bereits bekannten `MailItem`, `MeetingItem` usw. Der Zweck eines `StorageItem` ist, Einstellungen und ähnliche Benutzerdaten in Outlook zu speichern. Würde man solche Daten zum Beispiel in einer Excel-Arbeitsmappe oder einer Textdatei hinterlegen, müsste man diese Speicherdokumente manuell verwalten. Das bedeutet mehr Aufwand und kann auch zu Fehlern führen. Ein `StorageItem` hat demgegenüber den Vorteil, dass es in Outlook integriert ist. Wie die übrigen Elementtypen verwaltet Outlook auch die `StorageItem`-Objekte in Ordnern. Sie bleiben jedoch versteckt und sind in der Outlook-Bedienoberfläche im Ordner nicht sichtbar.

Ein `StorageItem` lässt sich über einen frei zu vergebenden Bezeichner identifizieren, den es in seinem Feld `Subject` speichert. Die Methode `GetStorage` des `Folder`-Objekts kann auf ein `StorageItem` zugreifen. Sie wird mit dem Bezeichner des `StorageItem` und einem weiteren Parameter aufgerufen, der die Art des Zugriffs kenntlich macht. Für den Zugriff per Bezeichner gibt man den Wert `olIdentifyBySubject` an. Wenn zu einem angegebenen Bezeichner noch kein `StorageItem` existiert, legt die Methode `GetStorage` es an.

Ein `StorageItem`-Objekt besitzt die Methoden `Save` und `Delete`. Die Methode `Save` speichert das `StorageItem` nach Änderungen. Die Methode `Delete` vernichtet es.

Ein `StorageItem` kann Informationen auf mehrere Arten speichern: als String in seinem Feld `Body`, als `UserProperty` oder als Anhang. Es besitzt eine Collection `UserProperties`, die mehrere `UserProperty`-Objekte aufnehmen kann. Die Speicherung als `UserProperty` eignet sich für einzelne Informationen wie Einstellungen, Vorgabewerte und Ähnliches und damit auch für die Demo-Anwendung „Dateianhänge-Auswahl v2“.

Wesentliche Felder des Objekttyps `UserProperty` sind `Name` und `Value`. Das Feld `Name` identifiziert das `UserProperty`. Das Feld `Value` speichert den eigentlichen Wert.

9.4.2 Code-Beispiel zum Outlook-Speicher

Der folgende Code-Schnipsel zeigt Möglichkeiten, um mit `StorageItem` und `UserProperties` umzugehen.

```

Sub StorageDemo()
Dim entwuerfe As Folder
Dim speicher As StorageItem
Dim info As UserProperty

Set entwuerfe = Application.Session.GetDefaultFolder(olFolderDrafts)
Set speicher = entwuerfe.GetStorage("Test", olIdentifyBySubject)
Debug.Print "Size: " & speicher.Size           ' Size: 0

Set info = speicher.UserProperties.Add("tzahl", olNumber)
info.Value = 9009
speicher.UserProperties.Add("ttext", olText).Value = "Zum Testen"
speicher.Save

With speicher
    Debug.Print .Subject & ": Anzahl " & .UserProperties.Count
                                ' Test: Anzahl 2
    Debug.Print .UserProperties(2).Name          ' tzahl
    Debug.Print .UserProperties("tzahl").Value   ' 9009
    Debug.Print .UserProperties("ttext").Value    ' Zum Testen
    .UserProperties("ttext").Value = "Geändert"
    Debug.Print .UserProperties("ttext").Value    ' Geändert
    ' Debug.Print .UserProperties("null").Value   ' Laufzeitfehler
End With
speicher.Delete
End Sub

```

Der Code-Schnipsel führt folgende Aktionen aus:

- Zunächst greift der Code mit `GetStorage` auf ein `StorageItem` „Test“ im Entwürfe-Ordner zu. Die Outlook-Konstante `olFolderDrafts` identifiziert dabei den Entwürfe-Ordner. Falls das `StorageItem` dort bisher noch nicht existiert, wird es angelegt und hat dann `Size=0`.
- Dann erzeugt der Code-Schnipsel ein `UserProperty`, das eine Zahl aufnimmt, und ein weiteres `UserProperty`, das einen Text aufnimmt, und speichert beide in das `StorageItem` „Test“. Dabei muss der Datentyp mit angegeben werden.
- Anschließend ruft der Code-Schnipsel die Werte der `UserProperty`-Objekte ab. Der Versuch, auf den Wert eines nicht vorhandenen `UserProperty`-Objekts zuzugreifen, würde einen Laufzeitfehler auslösen.
- Der Code-Schnipsel ändert den Wert eines `UserProperty` und druckt ihn aus.
- Zum Schluss löscht er das `StorageItem` „Test“.

Dieser Code-Schnipsel verwendet die Eigenschaft Session des Application-Objekts, um auf das NameSpace-Objekt zuzugreifen. Sie liefert dasselbe Ergebnis wie der Methodenaufruf GetNameSpace ("MAPI").

9.5 Demo-Anwendung „Dateianhänge-Auswahl v2“

In der zweiten Version der Demo-Anwendung „Dateianhänge-Auswahl v2“ können Benutzer den Pfad des Verzeichnisses mit Dateianhängen in einer UserForm konfigurieren. Der Verzeichnispfad wird im Outlook-Speicher gespeichert. Die Dateinamen der Anhänge bleiben weiterhin fest einprogrammiert. Dies ändert erst die dritte Version der Demo-Anwendung.

Gegenüber der ersten Version besitzt diese zweite Version der Demo-Anwendung eine zusätzliche UserForm AnhangKonfigForm und ein zusätzliches Code-Modul AnhangKonfig. Abb. 9.7 zeigt die Module im **Projekt-Explorer**. Die UserForm AnhaengeAuswahlForm und das Code-Modul AnhaengeAuswaehlen aus der ersten Version erfahren kleine Anpassungen.

9.5.1 Konfiguration des Verzeichnispfads per UserForm

Die neue UserForm AnhangKonfigForm enthält folgende Steuerelemente: ein Label mit einer erklärenden Beschriftung, ein Textfeld zum Anzeigen und Ändern des Dateipfads, einen **Speichern**-Button und einen **Abbrechen**-Button.

Der Code zur UserForm AnhangKonfigForm definiert das Verhalten der Steuerelemente:

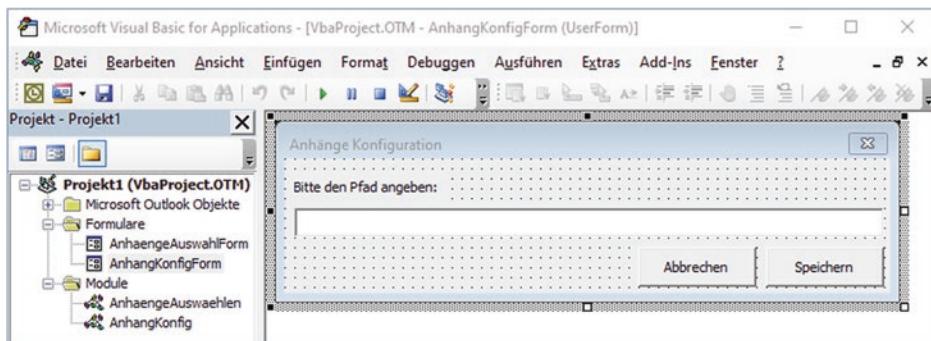


Abb. 9.7 UserForm zur Eingabe des Dateipfads für die Demo-Anwendung „Dateianhänge-Auswahl v2“

- Die Textbox zeigt den Pfad als String an, den sie aus einer Public-Variablen `pfad` übernimmt.
- Der **Speichern**-Button überträgt den möglicherweise geänderten Inhalt der Textbox in die Public-Variable `pfad` und schließt die UserForm.
- Der **Abbrechen**-Button bewirkt nichts außer die UserForm zu schließen.

Der Code zur UserForm `AnhangKonfigForm` besteht damit aus den folgenden drei kleinen Subs:

```
Private Sub UserForm_Initialize()
Me.TextBox1.Value = pfade
End Sub
```

```
Private Sub btnFertig_Click()
pfad = Me.TextBox1.Value
Me.Hide
End Sub
```

```
Private Sub btnCancel_Click()
Me.Hide
End Sub
```

Der folgende Abschnitt definiert den Code, der die UserForm `AnhangKonfigForm` öffnet.

9.5.2 VBA-Code des Moduls `AnhangKonfig`

Das neue Modul `AnhangKonfig` enthält den Code, der die Konfiguration des Verzeichnisfadls ermöglicht.

```
Public Const AKONFIG As String = "AnhangKonfig"
Public Const VPFAD As String = "AnhangKonfig"
Public pfade As String

Sub AnhangKonfigurieren()
Dim konfigForm As AnhangKonfigForm
Dim ordner As Folder
Dim konfig As StorageItem
Dim info As UserProperty

Set ordner = Application.Session.GetDefaultFolder(olFolderDrafts)
Set konfig = ordner.GetStorage(AKONFIG, olIdentifyBySubject)
```

```
If konfig.Size = 0 Then
    ' beim 1. Aufruf
    Set info = konfig.UserProperties.Add(VPFAD, olText)
Else
    Set info = konfig.UserProperties(VPFAD)
End If

' vorhandenen Wert lesen
pfad = info.Value

Set konfigForm = New AnhangKonfigForm
konfigForm.Show

' neuen Pfad speichern
info.Value = pfade
konfig.Save
End Sub
```

Das Modul AnhangKonfig enthält eine Public-Variable pfade, um den aktuellen Verzeichnispfad zwischenzuspeichern, und ein Makro AnhangKonfigurieren. Dieses Makro liest den Dateipfad aus der UserProperty eines StorageItem, überträgt ihn die Variable pfade und öffnet die UserForm AnhangKonfigForm. Die UserForm zeigt den Wert der Variable pfade und aktualisiert ihn, falls der Benutzer ihn in der UserForm editiert. Nachdem der Benutzer die UserForm wieder geschlossen hat, speichert das Makro den möglicherweise aktualisierten Wert von pfade wieder dauerhaft in das StorageItem.

Das StorageItem wird im Entwürfe-Ordner angelegt und bekommt ebenso wie das UserProperty die Benennung „AnhangKonfig“. Wenn das StorageItem „AnhangKonfig“ noch nicht existiert, also normalerweise beim ersten Aufruf, erzeugt das Makro das StorageItem und legt darin das UserProperty an.

Dieses Code-Beispiel folgt der guten Programmierpraxis, die Bezeichner von StorageItem und UserProperty in global sichtbaren Konstanten zu hinterlegen. So sind sie leicht zu finden, bei Bedarf leicht zu ändern und der Syntax-Checker der Entwicklungsumgebung kann auf Tippfehler beim Schreiben der Konstantenbezeichner hinweisen.

9.5.3 Anpassungen der UserForm AnhaengeAuswahlForm

Wenige Änderungen sind nötig, um die Verzeichnispfad-Konfiguration in die Demo-Anwendung zu integrieren. Die UserForm AnhaengeAuswahlForm erhält einen zusätzlichen Button Konfig wie in Abb. 9.8. Eine zusätzliche kleine Sub btnKonfig_Click

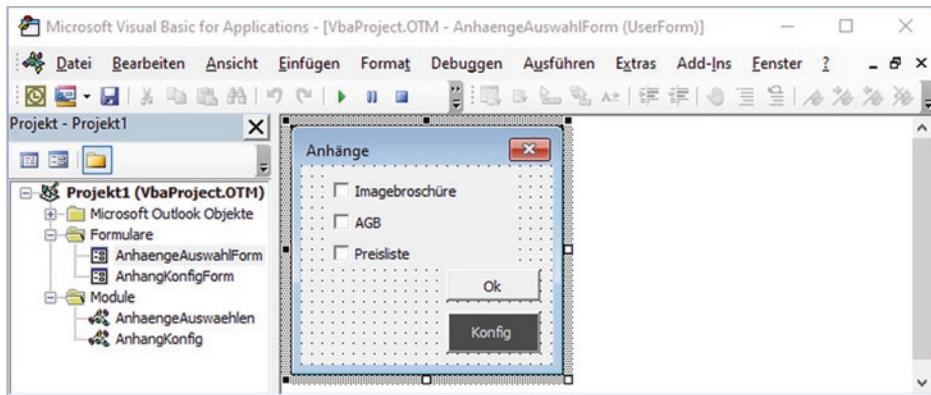


Abb. 9.8 UserForm AnhaengeAuswahlForm der Demo-Anwendung „Dateianhänge-Auswahl v2“

im Code-Modul der UserForm bewirkt, dass ein Klick auf diesen Button die Prozedur AnhangKonfigurieren aufruft.

```
Private Sub btnKonfig_Click()
    AnhangKonfigurieren
End Sub
```

Im nächsten Schritt wird die neu entwickelte Verzeichnispfad-Konfiguration weiter in die bestehende Demo-Anwendung integriert.

9.5.4 Modul AnhaengeAuswaehlen

Im Modul AnhaengeAuswaehlen wird die Konstante verzeichnispfad mit dem fest hinterlegten Pfad entfernt und durch eine Variable ersetzt, denn das Makro AnhangAuswaehlen ruft nun den Verzeichnispfad aus dem Outlook-Speicher ab. Die korrekten Bezeichner für StorageItem und UserProperty findet es zuverlässig in global sichtbaren Konstanten im Modul AnhangKonfigurieren. Der folgende Beispielcode zeigt das resultierende Modul AnhangAuswaehlen für die zweite Version der Demo-Anwendung.

```
' nicht mehr noetig in version 2:
' Const verzeichnispfad As String = "C:\Users\...\...\Anhaenge\"

' unverändert aus Version 1
Const anhang1 As String = "Imagebroschüre.pdf"
Const anhang2 As String = "AGB.pdf"
```

```
Const anhang3 As String = "Preisliste.pdf"

Public istAnhang1 As Boolean
Public istAnhang2 As Boolean
Public istAnhang3 As Boolean

Sub AnhangAuswaehlen ()
Dim auswahlForm As AnhaengeAuswahlForm
Dim nachricht As mailItem
' zwei neue Variablen in Version 2:
Dim speicher As StorageItem
Dim verzeichnispfad As String

' auf die Nachricht zugreifen, unverändert aus Version 1
If TypeName(ActiveWindow) = "Explorer" Then
    If Not ActiveExplorer.ActiveInlineResponse Is Nothing Then
        Set nachricht = ActiveExplorer.ActiveInlineResponse
    End If
ElseIf TypeName(ActiveWindow) = "Inspector" Then
    If TypeOf ActiveInspector.CurrentItem Is Outlook.mailItem Then
        Set nachricht = ActiveInspector.CurrentItem
    End If
End If
If nachricht Is Nothing Then      ' sollte nie vorkommen
    MsgBox ("Funktion Anhänge ist hier nicht möglich.")
    Exit Sub
End If

' neu in Version 2:
Set speicher = Session.GetDefaultFolder.olFolderDrafts) _
    .GetStorage(AKONFIG, olIdentifyBySubject)
If speicher.Size = 0 Then
    AnhangKonfigurieren
    Set speicher = Session.GetDefaultFolder.olFolderDrafts) _
        .GetStorage(AKONFIG, olIdentifyBySubject)
End If
verzeichnispfad = speicher.UserProperties(VPFAD).Value
If Right(verzeichnispfad, 1) <> "\" Then
    verzeichnispfad = verzeichnispfad & "\"
End If

' Auswahlform anzeigen, unverändert aus Version 1
Set auswahlForm = New AnhaengeAuswahlForm
auswahlForm.Show
```

```
If istAnhang1 Then
    nachricht.Attachments.Add verzeichnispfad & anhang1
End If
If istAnhang2 Then
    nachricht.Attachments.Add verzeichnispfad & anhang2
End If
If istAnhang3 Then
    nachricht.Attachments.Add verzeichnispfad & anhang3
End If
Set nachricht = Nothing
Set auswahlForm = Nothing
End Sub
```

In dieser Version „Dateianhänge-Auswahl v2“ ist der Verzeichnispfad flexibel konfigurierbar. Auch für die Dateinamen der drei Anhangdateien könnte man eine solche Konfigurationsmöglichkeit einrichten. Statt dies hier auszuprogrammieren, demonstriert nun die dritte Version der Demo-Anwendung eine interessantere, dynamische Lösung. Diese verwendet die Objektbibliothek MS Forms.

9.6 Demo-Anwendung „Dateianhänge-Auswahl v3“

Die dritte Version der Demo-Anwendung „Dateianhänge-Auswahl v3“ ist die komfortableste und flexibelste: Sie speichert wie die Version 2 den Pfad zum Dateiverzeichnis mit den Anhangdokumenten. Doch statt die Namen der Anhangdateien fest vorzugeben, liest sie die Namen der darin enthaltenen Dateien bei jedem Aufruf direkt aus dem Dateiverzeichnis und fügt sie zur UserForm hinzu. Dort angezeigte und im Verzeichnis tatsächlich vorhandene Dateien stimmen somit immer überein. Damit wird die Anwendung weniger fehleranfällig und sehr anpassungsfähig. Um eine weitere Datei als Anhang zur Verfügung zu stellen, legt man sie einfach in das Anhänge-Dateiverzeichnis. Genauso einfach kann man eine Datei aus der Auswahl nehmen, nämlich indem man sie aus dem Anhänge-Dateiverzeichnis entfernt.

In der zweiten Version der Demo-Anwendung konfiguriert der Benutzer den Dateipfad, indem er ihn manuell eintippt. Dies ist nicht nur ungewohnt und unkomfortabel, sondern auch fehlerträchtig. Die neue Version der Demo-Anwendung zeigt eine Alternative. Sie erlaubt dem Nutzer, einen Dateipfad festzulegen, indem er das gewünschte Dateiverzeichnis im Dateisystem ansteuert. Der Vorteil ist, dass so nur Pfade konfiguriert werden können, die im Dateisystem tatsächlich existieren. Leider beinhaltet das Objektmodell von Outlook keinen Dialog für interaktive Datei- oder Verzeichniswahl. Deshalb startet die Demo-Anwendung im Hintergrund Excel und verwendet den darin integrierten FileDialog aus dem Office-Objektmodell. Der Auswahldialog öffnet sich deswegen nur langsam. Man kann das tolerieren, wenn der Pfad nicht häufig neu konfiguriert werden muss.

9.6.1 Aufbau der Demo-Anwendung „Dateianhänge-Auswahl v3“

Diese Version der Demo-Anwendung benutzt Objekte aus mehreren externen Bibliotheken:

- das `FilesystemObject` aus der Microsoft Scripting Runtime, um die Dateien aus dem Verzeichnis zu lesen
- Steuerelemente aus MS Forms, um die UserForm zur Programmalaufzeit dynamisch aufzubauen
- den `FileDialog` aus dem Office-Objektmodell, um den Dateipfad zum Anhängeverzeichnis einzulesen.

Abb. 9.9 zeigt die Verweise auf die benötigten Objektbibliotheken. Bei den Komponenten der Anwendung gibt es gegenüber der Vorgängerversion folgende Änderungen:

- Die dynamische UserForm `AnhaengeDynForm` ersetzt die UserForm `AnhaengeAuswahlForm`.
- Das VBA-Code-Modul `AnhangKonfigPicker` mit dem Dateiverzeichnis-Auswahl-Dialog löst das bisher verwendete VBA-Modul `AnhangKonfig` ab.

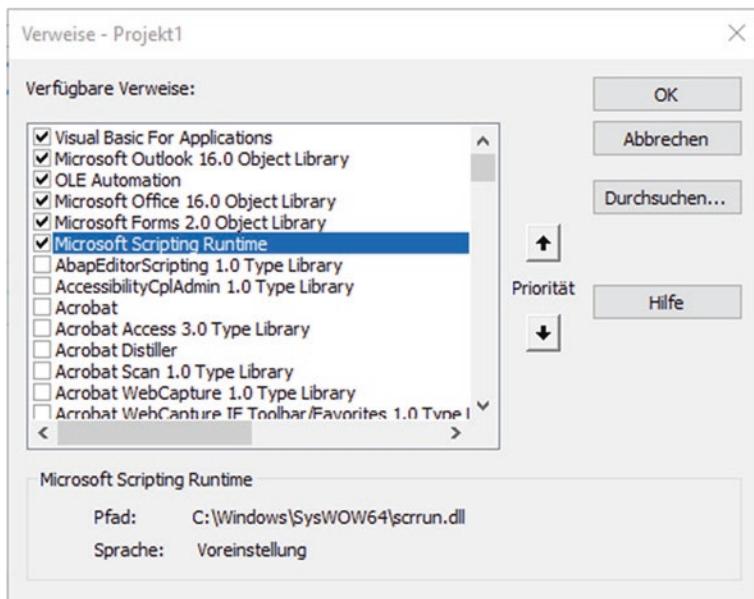


Abb. 9.9 Verweise auf externe Objektbibliotheken für die Demo-Anwendung „Dateianhänge-Auswahl v3“

- Die UserForm AnhangKonfigForm, die bisher die Eingabe des Dateipfads ermöglicht hat, fällt weg, da jetzt ein vorgefertigter Dialog zum Einsatz kommt.
- Das VBA-Code-Modul AnhaengeAuswaehlen erhält einige Anpassungen.

Damit besteht die Demo-Anwendung „Dateianhänge-Auswahl v3“ aus den Komponenten, die in Abb. 9.10 im **Projekt-Explorer** zu erkennen sind.

9.6.2 UserForm AnhaengeDynForm in der Entwurfsphase

Die UserForm zur Auswahl der Anhänge entsteht in zwei Schritten. In der Entwurfsphase wird eine unvollständige Vorversion der UserForm mit ihren unveränderlichen Bestandteilen definiert. Dies sind im Wesentlichen ein Label zur Anzeige des Verzeichnispfads sowie zwei Schaltflächen. Abb. 9.10 zeigt die UserForm so, wie sie in der Entwurfsphase eingerichtet wird. Die veränderlichen Bestandteile, nämlich eine Checkbox für jede Datei im Anhänge-Dateiordner, ergänzt die Sub UserForm_Initialize dynamisch zur Laufzeit. Sie läuft jedes Mal, wenn die UserForm geöffnet wird. Der Code der dynamisch erzeugten UserForm ist daher umfangreicher als der Code der statischen, komplett in der Entwurfsphase festgelegten UserForm.

9.6.3 Code der UserForm AnhaengeDynForm

Das Code-Modul der UserForm AnhaengeDynForm beinhaltet die Sub UserForm_Initialize mit zwei Hilfsprozeduren sowie zwei Prozeduren, die die Funktion der

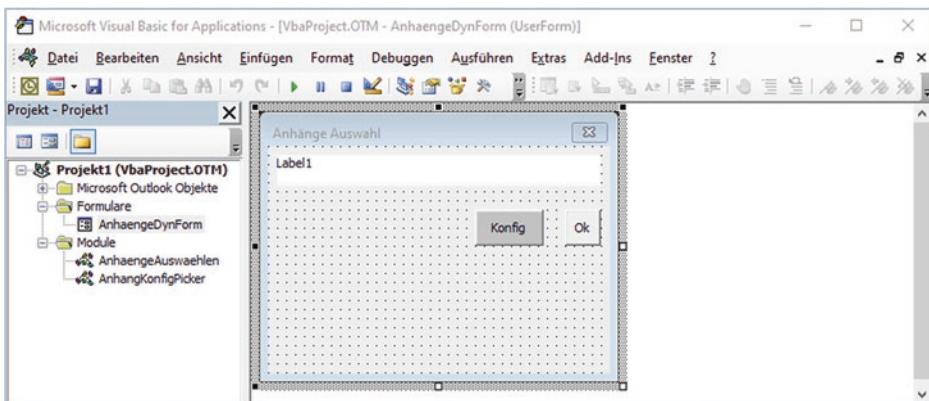


Abb. 9.10 Aufbau der Demo-Anwendung „Dateianhänge-Auswahl v3“ und vordefinierte Auswahl-UserForm

beiden Buttons realisieren. Es deklariert zudem zwei `Public`-Variablen, mit denen die UserForm Informationen mit dem Modul `AnhaengeAuswaehlen` austauscht.

9.6.3.1 Sub UserForm_Initialize der UserForm AnhaengeDynForm

Die `Sub UserForm_Initialize` komplettiert die UserForm zur Laufzeit. Sie führt bei jedem Öffnen der UserForm folgende Aktionen aus:

Zunächst entfernt sie alle CheckBoxes, die beim Öffnen eventuell schon in der UserForm vorhanden sind. Dies stellt sicher, dass die UserForm nur die aktuell im Verzeichnis vorhandenen Dateien anbietet. Der Code dafür liegt in der separaten Hilfsprozedur `CheckboxesEntfernen`.

Dann liest sie den konfigurierten Pfad zum Dateiverzeichnis aus dem `StorageItem` und schreibt ihn in das Label `lblPfad` der UserForm, um Benutzer zu informieren. Der Code zum Zugriff auf das `StorageItem` ist in eine eigene Funktion `PfadLesen` ausgelagert. Die `Sub UserForm_Initialize` ermittelt nun die Namen der Dateien im Anhänge-Dateiverzeichnis und erzeugt für jede Datei eine Checkbox. Die Bezeichner der Checkboxes setzt sie zusammen aus der Zeichenkette „`cbdat`“ und einer fortlaufenden Nummer. Zum Schluss passt die Sub die Positionen der beiden Buttons und die Höhe der UserForm für die jetzt vorhandene Anzahl von Checkboxen an.

```
Public verzeichnispfad As String
Public colDateien As New Collection

Private Sub UserForm_Initialize()
Dim FSO As New Scripting.FileSystemObject
Dim verzeichnis As Scripting.Folder
Dim dat As Scripting.File
Dim chkbox As MSForms.CheckBox
Dim i As Integer
Dim gap As Integer: gap = 18
Dim top_offset As Integer: top_offset = 20

CheckboxesEntfernen
verzeichnispfad = PfadLesen

On Error GoTo PfadNichtGefunden
' falls der konfigurierte Pfad nicht (mehr) existiert
Set verzeichnis = FSO.GetFolder(Me.verzeichnispfad)
On Error GoTo 0

Me.lblPfad.Caption = verzeichnispfad
i = 1
For Each dat In verzeichnis.Files
    Set chkbox = _
        
```

```

Me.Controls.Add("Forms.Checkbox.1", "cbdat" & i, True)
With chkbox
    .Top = i * gap + top_offset
    .Left = 20
    .Caption = dat.Name
    .Width = 190
    .Value = False
End With
i = i + 1
Next dat
btnOK.Top = (i + 1) * gap
btnKonfig.Top = (i + 1) * gap
Me.Height = btnOK.Top + btnOK.Height + gap + top_offset
Exit Sub

PfadNichtGefunden:
Me.lblPfad.Caption = verzeichnispfad _
& " - existiert nicht. Bitte konfigurieren!"
End Sub

```

Die Hilfsprozedur `CheckboxesEntfernen` läuft mit einer Schleife durch die Steuer-elemente der UserForm, entfernt alle Checkboxen und behält nur die vordefinierten Steuerelemente, also das Label und die beiden Buttons. Da ihre Namen immer mit „`cbdat`“ beginnen, sind die Checkboxes leicht zu erkennen.

```

Private Sub CheckboxesEntfernen()
Dim ctrl As MSForms.Control
For Each ctrl In Me.Controls
    If Left(ctrl.Name, 5) = "cbdat" Then
        Me.Controls.Remove ctrl.Name
    End If
Next ctrl
End Sub

```

Die Funktion `PfadLesen` arbeitet wie der entsprechende Code aus dem VBA-Code-Modul `AnhangKonfig` der Version 2 der Demo-Anwendung. Die Bezeichner des `StorageItem` und der `UserProperty` sind weiterhin als Konstanten `AKONFIG` be-ziehungsweise `VPFAD` im Modul `AnhangKonfigPicker` hinterlegt.

```

Private Function PfadLesen() As String
Dim speicher As StorageItem
Dim verzeichnispfad As String
Set speicher = Session.GetDefaultFolder(olFolderDrafts) _
    .GetStorage(AKONFIG, olIdentifyBySubject)

```

```

If speicher.Size = 0 Then
    AnhangKonfigurierenPicker
    ' jetzt ist ein Pfad im Speicher ...
    Set speicher = Session.GetDefaultFolder(olFolderDrafts) _
        .GetStorage(AKONFIG, olIdentifyBySubject)
End If

verzeichnispfad = speicher.UserProperties(VPFAD).Value
If Right(verzeichnispfad, 1) <> "\" Then
    verzeichnispfad = verzeichnispfad & "\"
End If

PfadLesen = verzeichnispfad
End Function

```

Die Sub UserForm_Initialize und ihre beiden Hilfsprozeduren Checkboxes-Entfernen und PfadLesen sind wie bereits gesagt im Code-Modul der UserForm AnhaengeDynForm definiert. Dazu kommen jetzt noch zwei Subs mit dem Code für die beiden Buttons der UserForm.

9.6.3.2 Code des „Ok“-Button mit Verwendung einer Collection

Die Sub btnOK_Click startet, wenn der Benutzer den „Ok“-Button klickt, also nachdem die benötigten Anhänge in der UserForm angehakt und damit die Werte der entsprechenden Checkboxes auf True gesetzt sind. Die Beschriftung einer Checkbox ist gleich dem Namen einer Anhang-Datei. Der Code muss also die Beschriftungen der angehakten Checkboxes sammeln, um die gewählten Anhänge zu ermitteln.

```

Private Sub btnOK_Click()
Dim ctrl As MSForms.Control
Dim cbox As MSForms.CheckBox

Set colDateien = Nothing           ' Collection leeren, siehe [2]
For Each ctrl In Me.Controls
    If Left(ctrl.Name, 5) = "cbdat" Then
        Set cbox = ctrl
        If cbox.Value Then
            colDateien.Add ctrl.Caption
        End If
    End If
Next ctrl
Me.Hide
End Sub

```

Bei der statischen Version der UserForm AnhaengeAuswahl war auch die Anzahl verfügbarer Anhänge fix vorgegeben. In der dynamischen Version ist sie dagegen veränderlich. Um eine veränderliche Anzahl von Informationen aufzunehmen, eignet sich besonders eine Collection. Die Methode btnOK_Click sammelt die Beschriftungen aller angehakten Checkboxen in einer Collection colDateien. Die Variable für die Collection ist im Code-Modul AnhaengeAuswaehlen als Public-Variable deklariert und damit auch im Code-Modul der UserForm AnhaengeDynForm sichtbar.

9.6.3.3 Code des „Konfig“-Button

Die Sub btnKonfig_Click läuft immer, wenn der Konfig-Button btnKonfig geklickt wird. Sie startet die Sub AnhangKonfigurierenPicker, die in einem eigenen Modul AnhangKonfigPicker liegt. Diese wiederum startet den FileDialog für die Verzeichnisauswahl. Weil der FileDialog langsam lädt, blendet die Sub btnKonfig_Click zuvor noch eine Info ein. Abschließend ruft sie nochmals die Sub UserForm_Initialize auf, damit die Dateiauswahl-UserForm AnhaengeDynForm ihre Anzeige aktualisiert und die Dateianhänge im jetzt neu konfigurierten Dateiverzeichnis anzeigt.

```
Private Sub btnKonfig_Click()
    Me.lblPfad = "Konfiguration startet gleich ..."
    AnhangKonfigurierenPicker    ' statt AnhangKonfigurieren
    UserForm_Initialize
End Sub
```

Damit ist das Code-Modul der UserForm AnhaengeDynForm komplett.

9.6.4 Modul AnhangKonfigPicker

Der Code im Modul AnhangKonfigPicker besteht aus zwei Konstantendefinitionen und der Sub AnhangKonfigurierenPicker, die die Konfiguration des Verzeichnispfads realisiert.

```
Public Const AKONFIG As String = "AnhangKonfig"
Public Const VPFAD As String = "AnhangKonfig"

Sub AnhangKonfigurierenPicker()
    Dim ordner As Folder
    Dim konfig As StorageItem
    Dim info As UserProperty
    Dim pfad As String
```

```
Dim xlApp As Object
Set xlApp = CreateObject("Excel.Application")
xlApp.Visible = False

Set ordner = Application.Session.GetDefaultFolder(olFolderDrafts)
Set konfig = ordner.GetStorage(AKONFIG, olIdentifyBySubject)

If konfig.Size = 0 Then
    ' beim 1. Aufruf
    Set info = konfig.UserProperties.Add(VPFAD, olText)
Else
    Set info = konfig.UserProperties(VPFAD)
End If

' vorhandenen Wert lesen
pfad = info.Value

Dim fd As Office.FileDialog
Set fd = xlApp.Application.FileDialog(msoFileDialogFolderPicker)

fd.AllowMultiSelect = False
If pfade <> vbNullString Then
    fd.InitialFileName = Left(pfad, InStrRev(pfad, "\"))
End If

Dim selectedItem As Variant
If fd.Show = -1 Then
    pfade = fd.SelectedItems(1)
End If

Set fd = Nothing
    xlApp.Quit
Set xlApp = Nothing

' neuen Pfad speichern
info.Value = pfade
konfig.Save
End Sub
```

Beim Start prüft die Sub, ob bereits ein Verzeichnispfad konfiguriert ist. Wenn ja, liest sie den Pfad aus dem StorageItem. Wenn nicht, erzeugt sie im StorageItem ein UserProperty für den Pfad. Sie konfiguriert den FileDialog für die Auswahl

des Dateiverzeichnisses so, dass dieser im Verzeichnis oberhalb des bisher konfigurierten Dateiverzeichnisses startet. So sehen User dieses Dateiverzeichnis im Verzeichnisystem und können sich leichter orientieren. Den Pfad zum übergeordneten Verzeichnis liefert der Ausdruck `Left(pfad, InStrRev(pfad, "\"))`. Zum Schluss, wenn der `FileDialog` wieder geschlossen ist, übernimmt die `Sub` den jetzt dort selektierten Pfad in das `UserProperty` und speichert das aktualisierte `StorageItem`.

9.6.5 Modul AnhaengeAuswaehlen v3

Im Vergleich zur Version 2 der Demo-Anwendung vereinfacht sich der Code im Modul `AnhaengeAuswaehlen`, weil jetzt der Code der `UserForm AnhaengeDynForm` mehr Aktionen übernimmt. Er ist auch deshalb kompakter, weil er die Bezeichner der selektierten Anhangdateien in einer einzigen Collection statt in mehreren einzelnen Variablen mit der `UserForm AnhaengeDynForm` austauscht.

```
Sub AnhangAuswaehlen()
Dim auswahlForm As AnhaengeDynForm
Dim nachricht As mailItem
Dim dat As Variant

' auf die Nachricht zugreifen
If TypeName(ActiveWindow) = "Explorer" Then
    If Not ActiveExplorer.ActiveInlineResponse Is Nothing Then
        Set nachricht = ActiveExplorer.ActiveInlineResponse
    End If
ElseIf TypeName(ActiveWindow) = "Inspector" Then
    If TypeOf ActiveInspector.CurrentItem Is Outlook.mailItem Then
        Set nachricht = ActiveInspector.CurrentItem
    End If
End If

If nachricht Is Nothing Then      ' sollte nie vorkommen
    MsgBox ("Funktion Anhänge ist hier nicht möglich.")
    Exit Sub
End If

' Auswahlform anzeigen
Set auswahlForm = New AnhaengeDynForm

auswahlForm.Show

For Each dat In auswahlForm.colDateien
```

```
nachricht.Attachments.Add auswahlForm.verzeichnispfad & dat  
Next dat  
End Sub
```

Die Public-Variable colDateien für die Collection und die Public-Variable verzeichnispfad für den Verzeichnispfad sind im Code der UserForm Anhaen-geDynForm deklariert (siehe Abschn. 9.6.3.1). Bezeichner des StorageItem und der UserProperty sind als Konstanten AKONFIG beziehungsweise VPFAD im Modul AnhangKonfigPicker hinterlegt.

9.7 Fazit

Es ist einfach, Makro-Aufrufe in die Bedienoberfläche von Office-Anwendungen einzubinden. Mit UserForms lassen sich Dialogfenster für die Dateneingabe und Informationsanzeige gestalten und auch zur Laufzeit durch VBA-Code dynamisch generieren. So können mit einfachen Mitteln komfortable Erweiterungen für Office-Anwendungen entstehen, die sich gut in bestehende Arbeitsabläufe einfügen. Dies wurde hier am Beispiel von MS Outlook gezeigt.

Literatur

1. o365devx, J. Mirabal, A. Jerabek, Office GSX, und K. Brandl, „Verwenden von Visual Basic for Applications in Outlook“, 4. Februar 2023. <https://learn.microsoft.com/de-de/office/vba/outlook/concepts/getting-started/using-visual-basic-for-applications-in-outlook> (zugegriffen 6. Februar 2023).
2. skkakkar, „vba – ideal way of clearing out collections – Stack Overflow“, *Stack Overflow*, 14. Juni 2016. <https://stackoverflow.com/questions/37820276/ideal-way-of-clearing-out-collections> (zugegriffen 2. März 2023).



Web-Dienste und REST APIs

10

Inhaltsverzeichnis

10.1	Nice to know: Web Services und REST APIs	182
10.1.1	Grundsätzliche Funktionsweise von Web Services	182
10.1.2	HTTP und HTTPS	182
10.1.2.1	URL	184
10.1.2.2	Request-Methoden	184
10.1.2.3	Header und Body	184
10.1.2.4	HTTP Statuscode	185
10.1.3	JSON für den Datenaustausch	185
10.1.4	APIs	188
10.2	Code-Beispiel „Postleitzahlen-API“ und JSON auswerten	189
10.2.1	Die Zippopotam.us-API	189
10.2.2	GET Request im Browser ausführen	189
10.2.3	Mit VBA HTTP GET Request senden und Response empfangen	190
10.2.4	JSON auswerten mit VBA-JSON	192
10.2.5	JSON auswerten mit MS Script Control	192
10.3	Tipps zum Umgang mit JSON in VBA	194
10.3.1	Umlaute in der HTTP Response	194
10.3.2	VBA-JSON und die Word-Objektbibliothek	194
10.4	Code-Beispiel „Spritpreise-Umkreissuche“ mit API-Key	195
10.5	Code-Beispiel „Stimmungserkennung“ mit POST Request	196
10.6	Fazit und Ausblick	197
	Literatur	198

In der Cloud stehen viele interessante Dienste als Web Services zur Verfügung, angefangen bei einfachen Wetterauskunftsdienssten über geschäftsrelevante Services von Logistik-, Bezahl- und anderen Dienstleistern bis hin zu innovativen Implementationen

Künstlicher Intelligenz. Geschäftssoftware für Customer Relationship Management, Enterprise Ressource Planning, Dokumentenmanagement etc. öffnet Schnittstellen, über die andere Softwaresysteme auf ihre Daten und Funktionen zugreifen können. Viele dieser Systeme werden als Server betrieben und bieten sogenannte REST APIs, die sich wie ein Web Service ansprechen lassen.

Die Vernetzung von Softwaresystemen mit Diensten ist ein wesentlicher Baustein der Digitalisierung. Auch VBA kann solche Dienste nutzen. Dies braucht nicht viel Programmieraufwand, denn auch dafür stehen Objektbibliotheken zur Verfügung. Dieses Kapitel demonstriert dies mit einfachen Code-Beispielen. Eine Einführung in das Thema Web Services und REST APIs schafft eine Basis, um die Code-Beispiele zu verstehen und für eigene Zwecke anzupassen.

10.1 Nice to know: Web Services und REST APIs

Ein Web Service (Dienst) wird von einem Server bereitgestellt und von einem Client genutzt. Client und Server sind zwei Softwaresysteme, die meist (aber nicht zwingend) auf zwei verschiedenen Rechnern im Netz laufen.

10.1.1 Grundsätzliche Funktionsweise von Web Services

Server bieten Dienste an. Sie stellen Informationen bereit, wie Wetterdaten, Wechselkurse, Preise und vieles mehr, oder führen Aktionen aus, zum Beispiel ein Ticket buchen, einen Social Media-Beitrag posten, eine Datenanalyse ausführen etc. Das Gegenstück des Servers ist der Client. Er nutzt die Dienste, indem er Wetterinfos, Währungskurse, Preise, News etc. abruft oder eine der genannten Aktionen anfordert. Server laufen ständig und warten auf Anfragen. Die Initiative zu einer Kommunikation ergreift der Client, indem er eine Anfrage sendet. Der Server liefert daraufhin eine Antwort. Für Anfrage und Antwort sind die technischen Begriffe „Request“ beziehungsweise „Response“ gebräuchig. Dies illustriert Abb. 10.1.

Die Dienste der Server sind darauf ausgelegt, dass beliebige Clients sie nutzen können. Eine mit VBA erstellte Anwendung übernimmt die Rolle des Clients. Mit welcher Technologie der Server realisiert ist, hat keine Bedeutung. Die technische Grundlage für die Kommunikation zwischen Client und Server sind HTTP und seine gesicherte Version HTTPS.

10.1.2 HTTP und HTTPS

HTTP (Hypertext Transfer Protocol) ist das Standardprotokoll des World Wide Web (WWW). Ein Protokoll ist dabei eine Vereinbarung darüber, wie der Client die Anfrage

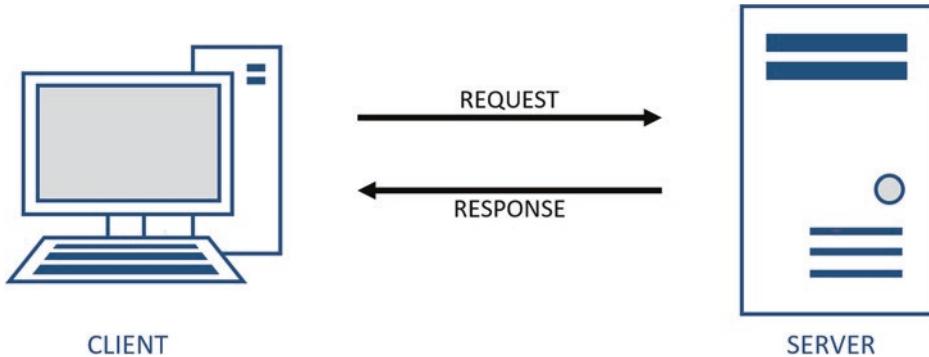


Abb. 10.1 Client und Server

und der Server die Antwort schickt. Es regelt, welche Nachrichten Client und Server in welcher Reihenfolge austauschen, welches Format die Nachrichten haben und was zwingend erforderliche Inhalte sind. HTTP wurde ursprünglich für die Auslieferung und Anzeige von Webseiten von einem Webserver zu einem Browser als Client entwickelt. Mittlerweile dient es auch für die Übertragung von Daten zwischen anderen Software-systemen. HTTP bildet eine Schicht in einem ganzen Stapel von Technologien und Protokollen, die es zusammen ermöglichen, dass Computer über das Internet miteinander kommunizieren.

Bekanntermaßen bringt der Datenaustausch über das Internet Risiken mit sich. Auch damit müssen sich die Nutzer solcher Technologien auseinandersetzen. HTTP überträgt Informationen als Klartext. Dies lässt Angreifern die Chance, diese Informationen mitzulesen und sich mit einer sogenannten Man-in-the-Middle-Attacke zwischen Client und Server dazwischenzuschalten. Dabei gibt sich ein Angreifer gegenüber dem Client als der eigentlich angesprochene Server aus, liest die Kommunikation unbemerkt mit und manipuliert sie.

Um HTTP abzusichern, kommt noch vor HTTP eine weitere Protokollsicht zum Stapel hinzu, nämlich der Secure Socket Layer (SSL). Die mit SSL abgesicherte Version von HTTP bezeichnet man als HTTPS. Bei HTTPS weist der Server dem Client seine Identität mit einem Zertifikat nach, und die übertragenen Informationen sind so verschlüsselt, dass nur die beteiligten Client und Server sie verstehen können.

Wenn wie in den Code-Beispielen dieses Kapitels keine vertraulichen Informationen übertragen und empfangene Daten lediglich wieder angezeigt werden, ist das Risiko, dass etwaige Angreifer einen Schaden verursachen, minimal. Wenn die vom Server gesendeten Daten jedoch in weitere Softwaresysteme, wie etwa ein Datenbanksystem, übergeben werden, lässt sich nicht völlig ausschließen, dass Angreifer Sicherheitslücken ausnutzen und mit einer Response auch Schadcode einschleusen, der diese Softwaresysteme manipuliert. Dann und auch wenn vertrauliche Daten kommuniziert werden, ist eine gesicherte Kommunikation per HTTPS eine unverzichtbare Schutzmaßnahme.

Für die VBA-Programmierung macht es keinen Unterschied, ob über HTTP oder über HTTPS kommuniziert wird, denn die zugrunde liegenden Technologien regeln die Absicherung und Verschlüsselung automatisch selbst. Wenn immer möglich verwendet man HTTPS. Jedoch stellen manche Anbieter ihre Dienste nur über HTTP bereit oder erheben Gebühren für den Zugriff über HTTPS.

Die folgenden Erläuterungen gelten gleichermaßen für HTTP und HTTPS, auch wenn der Einfachheit halber nur HTTP genannt wird.

10.1.2.1 URL

Eine Kommunikation über HTTP(S) startet, indem der Client einen Request an einen Server sendet. Ein wesentliches Element des Requests ist die URL. Die URL einer Anfrage an einen Postleitzahlen-Dienst könnte zum Beispiel so aussehen: [https://api.zippopotam.us/us/ca/Beverly Hills](https://api.zippopotam.us/us/ca/Beverly%20Hills). Sie beginnt mit der Angabe des Protokolls, hier „https“. Bei einer ungesicherten Verbindung würde die Protokollangabe „http“ lauten. Die URL setzt sich weiterhin zusammen aus der Web-Adresse des Dienstes, an den der Request gerichtet ist (hier „api.zippopotam.us“) und zusätzlichen Angaben und Parametern, die genau spezifizieren, was der Client vom Server anfordert, hier nämlich die Postleitzahl von Beverly Hills in Kalifornien, USA.

10.1.2.2 Request-Methoden

HTTP kennt verschiedene Formen von Requests, bezeichnet als Request-Methoden. Die meistgebrauchten Request-Methoden sind GET und POST. Wichtig ist hier: Der Server gibt die Methode vor und der Client muss den Request mit der Methode senden, die der Server erwartet. Sonst funktioniert die Kommunikation nicht.

10.1.2.3 Header und Body

HTTP Requests und HTTP Responses bestehen aus zwei Komponenten: dem Header und dem Body.

- Die Header von Request und Response transportieren Steuerungsinformation, zum Beispiel Login-Daten, mit denen sich der Client beim Server ausweist, oder Informationen über Art und Format der Inhalte, die Client und Server senden und empfangen. Sie werden in Form von Schlüssel-Wert-Paaren angegeben.
- Ein Request Body kann leer sein oder einen Inhalt haben. Das hängt von der Request-Methode ab. Bei der Request-Methode GET ist der Request Body immer leer. Wenn ein Request umfangreiche Einstellungen oder Daten zur Bearbeitung an den Server liefern muss, wird oft die Request-Methode POST verwendet. Dann kann der Client diese Daten im Request Body senden.
- In einer Response enthält der Response Body die Information, die der Client beim Dienst angefordert hat, zum Beispiel Wetterdaten, das Resultat einer Buchung, eine Postleitzahl etc.

10.1.2.4 HTTP Statuscode

Die HTTP Response beinhaltet einen Statuscode. Dieser ist eine dreistellige Nummer, die den Erfolg oder Misserfolg des Requests anzeigt. Ein Statuscode 200 oder eine andere Nummer aus dem Bereich 2xx besagt, dass der Server einen Request erfolgreich bearbeiten konnte. Ein Statuscode aus dem Bereich 4xx bedeutet einen Fehler in der Anfrage, etwa fehlerhafte Anfrageparameter, falsches Format, fehlende Zugriffs-berechtigung oder Ähnliches. Ein Statuscode aus dem Bereich 5xx signalisiert ein Problem des Servers: Der Server hat den Request zwar empfangen und auszuführen versucht, jedoch wurde dabei ein interner Fehler ausgelöst.

Tab. 10.1 listet als Beispiele einige häufig vorkommende Statuscodes auf. Statuscodes sind nicht standardisiert, sondern eine Konvention. Die Entwickler eines Services entscheiden darüber, welche Statuscodes ein Service in bestimmten Situationen sendet. Weil dabei auch ein gewisser Interpretationsspielraum bleibt, welcher der möglichen Statuscodes eine Situation am besten beschreibt, kann man sich auf die detaillierten Statuscodes nicht ganz verlassen. Die grobe Zuordnung 2xx = Erfolg, 4xx = Client-Fehler und 5xx = Server-Fehler passt aber fast immer.

10.1.3 JSON für den Datenaustausch

Client und Server müssen die Informationen, die sie miteinander austauschen, geeignet codieren, damit die jeweilige Gegenseite sie gut weiterverarbeiten kann. Zu diesem

Tab. 10.1 Die häufigsten HTTP Statuscodes [1]

Code		Bedeutung
200	Ok	Die Anfrage wurde erfolgreich bearbeitet
201	Created	Die Anfrage wurde erfolgreich bearbeitet und hat eine neue Ressource erstellt
204	No Content	Die Anfrage wurde erfolgreich durchgeführt, es gibt jedoch keine Antwortdaten zu senden
400	Bad request	Die Anfrage ist fehlerhaft
401	Unauthorized	Die Anfrage kann nicht ausgeführt werden, weil der Client nicht authentifiziert ist
403	Forbidden	Der Client ist zwar authentifiziert, hat aber keine Berechtigung für diese Anfrage
404	Not found	Die angeforderte Ressource existiert nicht oder wird versteckt
500	Internal Server Error	Der Server konnte die Anfrage wegen eines internen Fehlers nicht erfolgreich ausführen
501	Not Implemented	Der Server unterstützt diese Anfragemethode nicht
503	Service Unavailable	Der Server kann die Anfrage temporär nicht bearbeiten, zum Beispiel wegen Überlastung oder Wartungsarbeiten

Zweck wurden Datenaustauschformate entwickelt. Für Web Services sind XML und vor allem JSON gebräuchlich. Wer einen VBA-Client entwickelt, schreibt Code, der die richtigen Informationen aus einem empfangenen Request Body extrahiert, und möglicherweise auch Code, der den Request Body für die Anfrage erzeugt. Dabei sind dann Kenntnisse über das Datenaustauschformat gefordert.

JSON bedeutet „JavaScript Object Notation“, XML steht für „eXtensible Markup Language“. Obwohl in der Bezeichnung JSON der Begriff „JavaScript“ vorkommt, ist JSON genauso wie auch XML unabhängig von einer bestimmten Programmiersprache. XML ist älter und mächtiger, aber auch komplizierter als JSON. JSON braucht weniger Zeichen und ist für Menschen besser lesbar. Die meisten modernen APIs unterstützen inzwischen JSON. Auch dieses Kapitel konzentriert sich darauf.

HTTP transportiert Informationen grundsätzlich in Textform, daher sind auch JSON-Datenpakete im Request Body und Response Body immer Texte, die sich prinzipiell ohne spezielle Tools mit einem gewöhnlichen Editor anzeigen und bearbeiten lassen. Es gibt jedoch Tools, die den Umgang mit ihnen vereinfachen, zum Beispiel Webseiten wie [2], die einen JSON-String syntaktisch prüfen und ihn übersichtlich formatieren. Auch Code-Editoren wie VSCode und Notepad++ bieten spezielle Unterstützung für JSON. Ein JSON-Datenpaket repräsentiert immer ein Objekt. Das folgende Beispiel zeigt ein JSON-Objekt, das einer Variablen `Orte` zugewiesen ist:

```
Orte = {
    "country abbreviation": "DE",
    "places": [
        {
            "place name": "Rottweil",
            "longitude": "8.6422",
            "post code": "78628",
            "latitude": "48.1697"
        },
        {
            "place name": "Zimmern ob Rottweil",
            "longitude": "8.5576",
            "post code": "78658",
            "latitude": "48.1684"
        }
    ],
    "country": "Germany",
    "place name": "Rottweil",
    "state": "Baden-Württemberg",
    "state abbreviation": "BW"
}
```

Ein JSON-Objekt besteht aus elementaren Werten und weiteren JSON-Objekten, die auch in Listen organisiert sein können.

- Ein JSON-Objekt ist von einem Paar geschweifter Klammern umschlossen.
- Ein JSON-Objekt ist eine ungeordnete Menge von Name/Wert-Paaren. Ein Element des JSON-Objekts wird durch seinen Namen identifiziert.
- Zwischen zwei Name/Wert-Paaren steht ein Komma.
- Zwischen Name und Wert eines Name/Wert-Paares steht ein Doppelpunkt.
- Ein Wert ist ein elementares Datenobjekt (String, Zahl, Boolean oder Null-Wert), eine Liste oder selbst ein JSON-Objekt.
- Eine beliebige Anzahl von JSON-Objekten zwischen einem Paar eckiger Klammern bildet eine Liste.
- Eine Liste ist geordnet. Ein Element der Liste wird durch seine Position identifiziert.
- Alle JSON-Namen und JSON-Werte stehen in Anführungszeichen.

Auch im Beispiel oben ist das gesamte JSON-Objekt von geschweiften Klammern umschlossen. Es besteht aus sechs Elementen mit Namen "country abbreviation", "places", "country", "place name", "state" und "state abbreviation".

Außer "places" haben alle Elemente dieses JSON-Objekts elementare Datenobjekte als Wert, zum Beispiel das Name/Wert-Paar "country": "Germany" mit dem elementaren Wert "Germany". Der Wert zum Namen "places" ist dagegen eine Liste, die zwei JSON-Objekte enthält. Jedes davon besteht aus vier Name/Wert-Paaren mit elementaren Werten.

HTTP transportiert JSON-codierte Daten so wie auch alle anderen Daten in Textform, also als String. Um mit Programmcode bequem auf einzelne Datenelemente zugreifen zu können, analysiert man den JSON-String und baut daraus ein strukturiertes JSON-Objekt. Diesen Vorgang bezeichnet man als „Parsen“. Parsen ist aufwendiger als man vielleicht vermutet und sollte einer vorgefertigten Softwarebibliothek überlassen werden, die es in den meisten Programmiersprachen und auch für VBA gibt. In einem geparsten JSON-Objekt kann man mittels Pfaden auf einzelne Datenelemente zugreifen. Wie die Pfade im Programmcode genau anzugeben sind, hängen von der verwendeten Programmiersprache und Softwarebibliothek ab.

Für das oben gezeigte JSON-Objekt Orte ist beispielsweise der JSON-Pfad Orte.country sinnvoll. Er liefert den Wert "Germany". Die Zählung in Listen beginnt mit 0. Der JSON-Pfad Orte.places(1).place name liefert das Ergebnis „Zimmern ob Rottweil“.

10.1.4 APIs

Das Kürzel API steht für „Application Programming Interface“. Während ein „User Interface“ (UI) für Menschen gedacht ist, bietet ein „Application Programming Interface“ anderen Anwendungen (den „Applications“) Zugang zu den Funktionen eines Softwaresystems. Ergebnisse, die eine API liefert, sind nicht für menschliche Empfänger aufbereitet, sondern so, dass sie sich leicht maschinell verarbeiten lassen.

Eine API, die sich über das Protokoll HTTP ansprechen lässt, ist eine Web API. Als REST API oder RESTful API bezeichnet man APIs, wenn sie bestimmte Bedingungen erfüllen [3], die Details sind hier aber nicht relevant. REST ist kein Standard, sondern eher ein Programmierstil. Er ist inzwischen so verbreitet, dass die meisten modernen Web APIs als REST APIs gestaltet sind.

Die API des Dienstes spezifiziert, welche Form Anfragen haben müssen und welche Form das zurückgelieferte Ergebnis hat. Während das Protokoll HTTP standardisiert ist, können Entwickler von REST APIs diese frei gestalten. Zu einer API gibt es daher meist auch eine API-Dokumentation, die den Aufbau der Requests und Responses und das Format der Request-URL beschreibt. Oft zeigen auch Beispiele, wie verschiedene Tools und Programmiersprachen die API ansprechen können. Für die Programmierung eines Clients muss man die API des Dienstes kennen und bei komplexen APIs manchmal auch mit verschiedenen Aufruf-Varianten experimentieren, bis es gelingt, erfolgreiche Requests zu senden.

Abb. 10.2 gibt einen Überblick über die wichtigsten der bisher genannten Begriffe und zeigt, wie sie zusammenspielen. Die folgenden Abschnitte zeigen Code-Beispiele mit VBA.

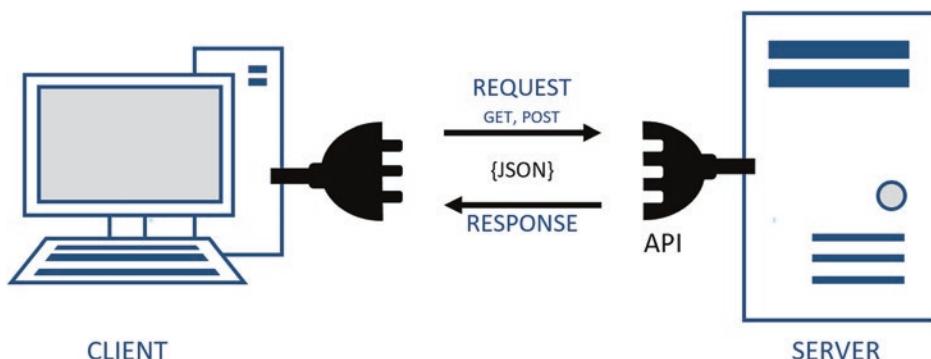


Abb. 10.2 REST API

10.2 Code-Beispiel „Postleitzahlen-API“ und JSON auswerten

Dieses Code-Beispiel greift auf die API von Zippopotam.us zu. [4] Zippopotam.us ist ein Webdienst, der Postleitzahlen und geografische Informationen zu Orten liefert. Er ist frei zugänglich, verlangt keinen Login und hat eine einfache API. Das Beispiel-JSON-Objekt aus Abschn. 10.1.3 ist das Ergebnis eines Requests an Zippopotam.us.

10.2.1 Die Zippopotam.us-API

Die API von Zippopotam.us verarbeitet GET Requests. Alle Informationen, die der Dienst zur Ausführung einer Anfrage braucht, kann der Client in die URL codieren. Die API von Zippopotam.us akzeptiert zwei Eingabevarianten: Sendet der Client einen Staat plus Ortsbezeichnung, dann liefert Zippopotam.us die dazugehörigen Postleitzahlen. Sendet der Client eine Postleitzahl, liefert der Dienst den entsprechenden Ort. Das JSON-Beispiel in Abschn. 10.1.3 ist die Antwort auf den Request <https://api.zippopotam.us/de/bw/Rottweil>. Tab. 10.2 zeigt weitere Beispiele: links für den Abruf von Orten zur Postleitzahl und rechts für den Abruf der Postleitzahlen zum Ort.

10.2.2 GET Request im Browser ausführen

Einen GET Request kann auch ein Browser ausführen. Wenn man zum Beispiel mit dem Browser Firefox die URL <https://api.zippopotam.us/de/bw/Rottweil> öffnet und dadurch die API von Zippopotam.us anspricht, erhält man den Response Body, der in Abb. 10.3 zu sehen ist. In der Abbildung ist die Ansicht **JSON** geöffnet, die den JSON-String aufbereitet anzeigt. In der Ansicht **Rohdaten** erscheint er als Zeichenkette ohne Einrückungen und Zeilenumbrüche. In der Ansicht **Kopfzeilen** zeigt der Browser den Response Header sowie den Request Header, den er automatisch an die API mitgeschickt hat. Auch ein VBA-Client empfängt diese Response, wenn er die API anspricht. Er kann sie dann auswerten und mit den darin enthaltenen Informationen weiterarbeiten.

Tab. 10.2 Beispiele für Requests an api.zippopotam.us

Abruf mit Postleitzahl	Abruf mit Staat plus Ortsbezeichnung
https://api.zippopotam.us/us/90210	https://api.zippopotam.us/us/ca/Beverly Hills
https://api.zippopotam.us/de/70734	https://api.zippopotam.us/de/bw/Fellbach
https://api.zippopotam.us/de/81541	https://api.zippopotam.us/de/by/München

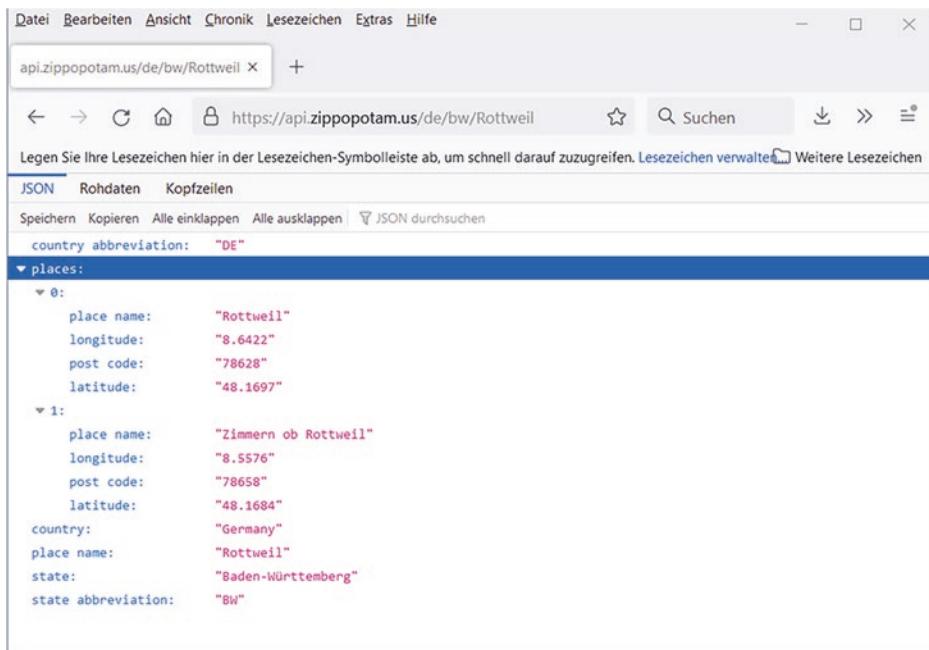


Abb. 10.3 JSON-Ausgabe eines GET Requests im Browser

10.2.3 Mit VBA HTTP GET Request senden und Response empfangen

Der folgende Code-Schnipsel implementiert einen Client in einer Excel-Arbeitsmappe. Er verwendet dazu das Objekt `ServerXMLHTTP60` aus der VBA-Objektbibliothek MSXML. Abb. 10.4 zeigt die Verweise auf diese Objektbibliothek und auf einige weitere Bibliotheken, die in den folgenden Abschnitten verwendet werden.

Im Code-Beispiel sendet das `ServerXMLHTTP60`-Objekt einen HTTP GET Request an die Zippopotam.us-API und nimmt die HTTP Response entgegen. Der unveränderliche Teil der URL ist als Konstante deklariert. Die wechselnden Postleitzahlen erfragt der Code mit einer InputBox und erstellt damit URLs wie in Tab. 10.2. Das Länderkürzel „DE“ ist hier im Code fest vorgegeben.

```

Const ZIP_URL As String = "https://api.zippopotam.us/DE/"

Sub zipcode()
  ' Einfacher GET Request ohne Authentifizierung

```

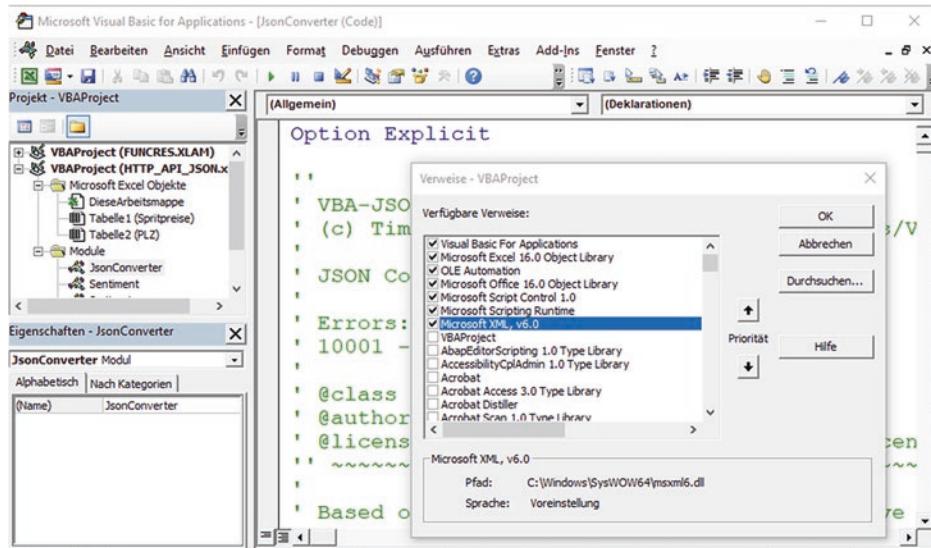


Abb. 10.4 Objektbibliotheken für Web Clients

```

Dim zip As String
Dim req As String

Dim XMLHttpRequest As New MSXML2.ServerXMLHTTP60

zip = InputBox("Bitte eine PLZ in Deutschland eingeben:")
req = ZIP_URL & zip

' Request absenden
XMLHttpRequest.Open "GET", req, False
XMLHttpRequest.send

' Rückgabewerte ausgeben
MsgBox "Status: " & XMLHttpRequest.Status
MsgBox "responseText: " & XMLHttpRequest.responseText
If (XMLHttpRequest.Status = 200) Then
    JsonAnalysieren (XMLHttpRequest.responseText)
End If
End Sub

```

Der Client zeigt den empfangenen Statuscode und die Response des Dienstes in je einer MessageBox an. Im Erfolgsfall, also bei Statuscode 200, analysiert er die JSON-Antwort, die der Server zurückliefert. Die JSON-Analyse erläutert der folgende Abschnitt.

10.2.4 JSON auswerten mit VBA-JSON

Die Bibliothek VBA-JSON [5] analysiert JSON-Strings und baut daraus Datenstrukturen, die die einzelnen Elemente über JSON-Pfade zugänglich machen. VBA-JSON steht als VBA-Quellcode auf Github zur Verfügung. Um sie in ein VBA-Projekt einzubinden, muss der Quellcode in einem Modul namens `JsonConverter` liegen. Sehr einfach erstellt man dieses Modul, indem man die Datei „`JsonConverter.bas`“ [6] von Github herunterlädt und mit der Import-Funktion der VBA-Entwicklungsumgebung (**Datei > Datei importieren ...**) einliest. Man kann den Code auch per Copy&Paste in ein Modul einfügen, das man „`JsonConverter`“ nennt. VBA-JSON baut die Objektstruktur eines JSON-Objekts mit dem Objekttyp `Dictionary` auf, den die Objektbibliothek Microsoft Scripting Runtime liefert. Deshalb muss die Objektbibliothek Microsoft Scripting Runtime eingebunden werden, siehe Abb. 10.4. In der Abbildung ist auch das Modul `JsonConverter` mit dem Quellcode von VBA-JSON zu erkennen.

Der `JsonConverter` stellt die Methode `ParseJson` bereit. Sie transformiert einen JSON-String in eine aus `Dictionary`-Objekten bestehende Objektstruktur, die sich leicht auswerten lässt. Im Code-Beispiel erhält der `JsonConverter` den `ResponseText` aus der HTTP Response des API-Aufrufs. Dieser `ResponseText` hat das in Abb. 10.3 gezeigte Format. Das Ergebnis des Parsens speichert der Code in die Objektvariable `responseObjekt`. Die genaue Struktur dieses Objekts ist an sich nicht relevant, wichtig ist lediglich, dass daraus mittels JSON-Pfaden gezielt einzelne Informationen abgerufen werden können. Das Code-Beispiel zeigt, wie diese JSON-Pfade für VBA-JSON zu schreiben sind, um auf einzelne Inhalte des JSON-Objekts zuzugreifen. Die Zählung in Listen beginnt hier mit 1, abweichend von vielen anderen Programmiersprachen und -situationen.

```
Sub JsonAnalysieren(responseText As String)
Dim responseObjekt As Object
Set responseObjekt = JsonConverter.ParseJson(responseText)

Cells(2, 2).Value = responseObjekt("post code")
Cells(4, 2).Value = responseObjekt("places")(1)("place name")
Cells(5, 2).Value = responseObjekt("places")(1)("state")
Cells(6, 2).Value = responseObjekt("country")
End Sub
```

10.2.5 JSON auswerten mit MS Script Control

Eine Alternative zu VBA-JSON bietet das `ScriptControl`-Objekt aus der Objektbibliothek MS Script Control, deren Referenzierung in Abb. 10.4 ebenfalls zu sehen ist.

Die ScriptControl ist ein JScript-Interpreter, der JavaScript ausführen und JSON parsen kann. Dies zeigt das folgende Code-Beispiel.

```
Sub JsonAnalysieren2(responseText As String)
Dim scrContr As New MSScriptControl.ScriptControl
scrContr.Language = "JScript"

Dim responseObjekt As Object
Dim place1Objekt As Object
Set responseObjekt = scrContr.Eval("(" & responseText & ")")
Set place1Objekt = CallByName(responseObjekt.places, 0, VbGet)

Cells(2, 2).Value = CallByName(responseObjekt, "post code", VbGet)
Cells(4, 2).Value = CallByName(place1Objekt, "place name", VbGet)
Cells(5, 2).Value = CallByName(place1Objekt, "state", VbGet)
Cells(6, 2).Value = CallByName(responseObjekt, "country", VbGet)
End Sub
```

Auch die ScriptControl erhält den ResponseText aus der HTTP Response des API-Aufrufs. Sie wertet ihn mit ihrer Methode Eval aus und speichert das Ergebnis des Parsens in die Objektvariable responseObjekt. Wie das Code-Beispiel zeigt, ist der Zugriff auf einzelne Elemente von responseObjekt hier komplexer als mit der Objektbibliothek VBA-JSON.

Um auf ein bestimmtes Element von responseObjekt zuzugreifen, setzt das Code-Beispiel die Funktion CallByName ein. Das erste Argument von CallByName ist ein Objekt und das zweite Argument ist eine Methode dieses Objekts. Das dritte Argument ist die in VBA vordefinierte Konstante VbGet. Sie signalisiert, dass die Methode einen Lesezugriff ausführt.

Der erste Einsatz von CallByName speichert ein Teillokett des responseObjekt in einer zweiten Objektvariablen, nämlich das erste Element der Liste places in die Objektvariable place1Objekt. Die Zählung in Listen beginnt hier bei 0. Die weiteren Verwendungen von CallByName speichern Elemente von place1Objekt und von responseObjekt in Zellen auf einem Excel-Arbeitsblatt.

Der Vorteil der MS ScriptControl gegenüber VBA-JSON ist, dass es sich dabei um eine zum System gehörende Objektbibliothek handelt. Allerdings führt die ScriptControl-Methode Eval beliebigen JavaScript-Code aus und öffnet damit eine ernst zunehmende Sicherheitslücke, denn eine böswillige oder von einem Angreifer gekaperte API könnte in einem responseObjekt bösartigen JavaScript-Code einschleusen. Daher darf die MS Script Control nur mit bekannten, zuverlässigen APIs verwendet werden [7, 8]. Eine weitere Möglichkeit, um JSON mit VBA zu parsen, wäre, einen JSON-Parser selbst zu programmieren. Eine Anleitung gibt [9].

10.3 Tipps zum Umgang mit JSON in VBA

Bei der Verarbeitung von JSON mit VBA treten gelegentlich kleine Probleme auf. Die folgenden Abschnitte bieten Lösungen.

10.3.1 Umlaute in der HTTP Response

Umlaute und andere Sonderzeichen liefert die HTTP Response in codierter Form, zum Beispiel „ü“ als „\u00f6“, siehe Abb. 10.5. Excel und Word wandeln die Codes automatisch in Umlaute um, wie in Abb. 10.5 ebenfalls zu sehen ist. Wenn man die Response außerhalb von Excel & Co. weiterverarbeiten möchte, kann man die Codes konvertieren, zum Beispiel mit einer Funktion, die D. Gorelenkov entwickelt hat [10].

10.3.2 VBA-JSON und die Word-Objektbibliothek

VBA-JSON verwendet den Objekttyp `Dictionary` aus der Objektbibliothek Microsoft Scripting Runtime. Verwendet man VBA-JSON in Word, führt dies zu einem Typkonflikt, denn das Objektmodell von Word enthält ebenfalls einen Objekttyp `Dictionary`. Das Problem lässt sich beheben, indem man den Quellcode von VBA-JSON etwas modifiziert und dort `Dictionary` zu `Scripting.Dictionary` ergänzt, denn damit ist der Typkonflikt aufgelöst.

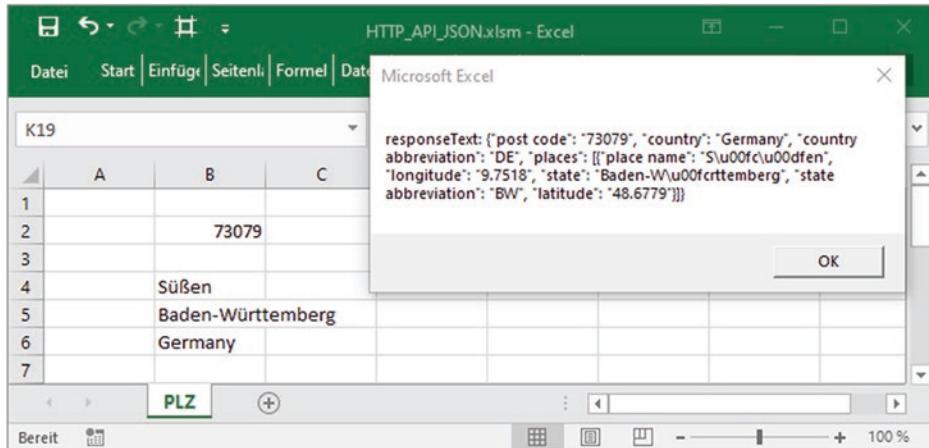


Abb. 10.5 Codierte Umlaute in einer HTTP Response

10.4 Code-Beispiel „Spritpreise-Umkreissuche“ mit API-Key

Viele Web-APIs verlangen einen API-Key als Login-Password oder Zugangsschlüssel. Ein Dienst kann durch den API-Key den anfragenden Client identifizieren, Berechtigungen steuern und Anfragen begrenzen oder zählen, um sie in Rechnung stellen.

Das folgende Code-Beispiel spricht den Webservice „Umkreissuche“ von Tankerkönig [11] an, der aktuelle Preise für Benzin, Super und Diesel von Tankstellen in einer bestimmten Region ausgibt. Dieser Service verlangt einen API-Key. Auf der Webseite des Tankerkönigs findet man einen Demo-API-Key für Tests oder kann einen eigenen API-Key kostenlos anfordern. Um das Code-Beispiel auszuführen, muss im Code ein gültiger API-Key eingefügt werden. Dabei ist zu bedenken, dass ein API-Key wie ein Passwort funktioniert und genauso sorgfältig behandelt und geschützt werden muss.

Auch dieses Code-Beispiel sendet einen GET Request. Ein Client, der die Umkreissuche von Tankerkönig nutzen möchte, muss die in Abb. 10.6 aufgeführten Parameter angeben und zu einer URL wie der folgenden zusammenbauen:

<https://creativecommons.tankerkoenig.de/json/list.php?lat=48.1697&lng=8.6422&rad=1.5&sort=dist&type=diesel&apikey=00000000-0000-0000-0000-00000000000x>

Die Prozedur `umkreissuche` des Code-Beispiels liest alle Parameter außer dem API-Key aus einem Excel-Arbeitsblatt „Spritpreise“ wie in Abb. 10.6. Sensible Informationen wie ein API-Key sind in einer Konstanten im Code besser aufgehoben. Längen- und Breitengrad zu einem Ort könnte der Client beispielsweise auch von Zippopotam.us erfragen.

```
Const SPIT_URL = _
    "https://creativecommons.tankerkoenig.de/json/list.php?"
Const APIKEY = ""           ' hier einsetzen

Sub umkreissuche()
```

A	B	C	D
1 Parameter	Bedeutung	Eingabe	Anmerkung
2 lat	geographische Breite	48.0124	
3 lng	geographische Länge	8.5551	
4 rad	Suchkreis in km	5	Maximaler Wert: 25km
5 type	Spritsorte	diesel	Mögliche Werte: e5, e10, diesel oder all
6 sort	Sortierung	price	Mögliche Werte: price, dist
7			
8			
	Spritpreise		
	Bereit		

Abb. 10.6 Parameter der API Umkreissuche von Tankerkönig

```
Dim req As String
Dim param As Integer
Dim XMLHttp As New MSXML2.ServerXMLHTTP60

req = SPRIT_URL
With Worksheets("Spritpreise")
    For param = 2 To 6
        req = req & Cells(param, 1).Value & "=" _
            & Cells(param, 3).Value & "&"
    Next param
    req = req & "apikey=" & APIKEY
End With

XMLHttp.Open "GET", req, False
XMLHttp.send
If (XMLHttp.Status = 200) Then
    MsgBox "Status: " & XMLHttp.Status
    MsgBox "responseText: " & XMLHttp.responseText
End If
End Sub
```

10.5 Code-Beispiel „Stimmungserkennung“ mit POST Request

POST Requests kommen typischerweise zum Einsatz, wenn komplexere Aufgaben auszuführen sind oder wenn Clients umfangreiche Datenpakete an einen Dienst senden. APIs, die mit POST Requests arbeiten, sind daher oft aufwendiger anzusprechen als APIs mit GET Request, und werden auch seltener frei angeboten. Das folgende Code-Beispiel demonstriert einen POST Request mithilfe der Sentiment API, die sehr einfach zu nutzen ist. Sie beruht auf Künstlicher Intelligenz und analysiert mehr oder weniger erfolgreich die emotionale Stimmung, die sich in einem Text widerspiegelt [12]. Wer eingehende E-Mails, Social Media-Beiträge und ähnliche Texte automatisch bearbeitet, möchte vielleicht einen solchen Dienst nutzen, um Stimmungstrends zu erkennen und in der Antwort den richtigen Ton zu treffen. Auch ausgehende Nachrichten und selbst verfasste Texte könnte man damit prüfen, um Formulierungen mit unbeabsichtigt negativer Wirkung zu erkennen und zu vermeiden.

Anders als ein GET Request sendet ein POST Request einen Request Body. Bei der Sentiment API enthält der Request Body den Text, der analysiert werden soll. Den API-Key und eine weitere, der Steuerung dienende Information sendet das Code-Beispiel im Request Header.

```
Const APIKEY As String = "XXXXXXXXXXXXXXXXXXXXXX"      ' hier ersetzen
Const SENT_URL As String = _
    "https://api.apilayer.com/sentiment/analysis"

Sub SentimentPost()
Dim XMLHttp As New MSXML2.ServerXMLHTTP60
Dim requestbody As String

requestbody = "Alles hat super geklappt. Das ist ein tolles" _
& " Produkt. Danke! Ich freue mich sehr."

XMLHttp.Open "POST", SENT_URL, False

XMLHttp.setRequestHeader "apikey", APIKEY
XMLHttp.setRequestHeader "Content-Type", "text/plain"

XMLHttp.send requestbody

MsgBox "Status: " & XMLHttp.Status
MsgBox "responseText: " & XMLHttp.responseText
End Sub
```

Als Response liefert der Sentiment-Dienst einen JSON-String wie diesen:

```
{
    "sentiment": "positive",
    "score": 4,
    "confidence": 89.99999761581421,
    "language": "de",
    "content_type": "text"
}
```

Die Sentiment API steht auf APILayer zur Verfügung. Diese Plattform erlaubt Entwicklern, REST APIs anzubieten und zu vermarkten. Die APIs sind einheitlich und gut dokumentiert und viele lassen sich kostenlos testen. Ein API-Key ist erforderlich, aber mit wenig Aufwand zu erhalten [13].

10.6 Fazit und Ausblick

Web Services und APIs können Anwendungen unaufwendig mit Künstlicher Intelligenz ausstatten und mit Informationen oder Medieninhalten aus externen Quellen anreichern. Ihre Nutzung erfordert jedoch Sorgfalt und Überlegung. Datenschutz, Geheim-

haltungspflichten, rechtliche Vorschriften zur Datenverarbeitung und auch die Nutzungsbedingungen der Dienstanbieter müssen eingehalten werden.

Viele professionelle Dienstleister ermöglichen es heute, ihre Dienste über APIs in andere Software zu integrieren. Drei von vielen Beispielen sind etwa [14]–[16]. Business-Softwaresysteme für Enterprise Ressource Planning, Contentmanagement, Customer Relationship Management, Dokumentenmanagement, Product Lifecycle Management usw. müssen vielen Mitarbeitern zur Verfügung stehen und werden schon deswegen zentral als Server betrieben. Hier bietet es sich an, Funktionen auch über Rest APIs zur Verfügung zu stellen, und es ist heute fast Standard, dass die Systeme damit ausgestattet sind. Wenn die Unternehmens-IT entsprechende Zugriffsrechte gewährt, können auch VBA-Tools diese APIs nutzen, um oft mühselige und zeitaufwendige Dateneingaben und -abrufe zu automatisieren.

Die hier gezeigten Code-Beispiele bieten nur einen ersten Einstieg in die Nutzung von REST APIs mit VBA. Wegen der hohen Sicherheitsanforderungen sind gerade REST APIs von Unternehmenssoftware manchmal schwer anzusprechen. Recherchen und Experimente sind dann nötig, um das genaue Format für funktionierende Requests herauszufinden. Doch die Vorteile des automatischen Datenaustauschs und der schnellen Bereitstellung von besserer und umfangreicherer Information wiegen diesen Aufwand auf.

Literatur

1. MDN contributors, „HTTP response status codes – HTTP | MDN“, *mdn web docs*, 3. Juli 2022. <https://developer.mozilla.org/en-US/docs/Web/HTTP>Status> (zugegriffen 7. September 2022).
2. M. Drapeau *u. a.*, „JSON Beautifier – CSVJSON“, *JSONBeautifier*, 2022. https://csvjson.com/json_beautifier (zugegriffen 2. März 2023).
3. L. Gupta, „What is REST“, *REST API Tutorial*, 7. April 2022. <https://restfulapi.net/> (zugegriffen 9. September 2022).
4. T. Hauet, „Zippopotamus- Zip Code Galore“, 17. Dezember 2013. <https://zippopotam.us/> (zugegriffen 10. September 2022).
5. T. Hall, S. Liu, Red Stapler, und P. Swannell, „VBA-JSON“. VBA-tools, 28. Januar 2019. <https://github.com/VBA-tools/VBA-JSON> (zugegriffen 7. September 2022).
6. T. Hall, „JsonConverter.bas“, VBA-JSON, 28. Januar 2019. <https://raw.githubusercontent.com/VBA-tools/VBA-JSON/master/JsonConverter.bas> (zugegriffen 7. September 2022).
7. MrExcel, „VBA macro to extract web content“, *MrExcel Message Board*, 13. Juli 2020. <https://www.mrexcel.com/board/threads/vba-macro-to-extract-web-content.1139807/> (zugegriffen 14. September 2022).
8. The Excel Development Platform, „The Excel Development Platform: VBA – Parse JSON safer with JSON.Parse and not Eval“, *The Excel Development Platform*, 16. Januar 2018. <https://exceldevelopmentplatform.blogspot.com/2018/01/vba-parse-json-safer-with-jsonparse-and.html> (zugegriffen 14. September 2022).

9. D. Ferry, „Excel VBA: Parse JSON, Easily“, *The Startup*, 27. April 2021. <https://medium.com/swlh/excel-vba-parse-json-easily-c2213f4d8e7a> (zugegriffen 14. September 2022).
10. D. Gorelenkov, „Function to unescape UTF-8 escaped content for VBA“, *Gist*. <https://gist.github.com/yadimon/ce1d04b88de17064bfae> (zugegriffen 18. September 2022).
11. Tankerkönig, „Tankerkönig API“. <https://creativecommons.tankerkoenig.de/> (zugegriffen 13. September 2022).
12. APILayer, „Sentiment Analysis API“, *ApiLayer*, 2022. <https://apilayer.com/marketplace/sentiment-api> (zugegriffen 10. September 2022).
13. APILayer, „Managing API Keys“, *APIlayer | Hassle-free API marketplace*, 2022. <https://api-layer.com/docs/article/managing-api-keys> (zugegriffen 19. September 2022).
14. AutoScout24, „Die AutoScout24 API“, *AutoScout24*. <https://www.autoscout24.de/haendler-portal/api/> (zugegriffen 25. Februar 2023).
15. PayPal, „Get started with PayPal Developer“. <https://developer.paypal.com/api/rest/> (zugegriffen 25. Februar 2023).
16. DHL, „DHL Group API Developer Portal | DHL API Developer Portal“. <https://developer.dhl.com/> (zugegriffen 25. Februar 2023).



Inhaltsverzeichnis

11.1	Objektmodell der SAP GUI	202
11.1.1	Objekte des SAP GUI-Objektmodells	203
11.1.2	Aufbau eines SAP GUI-Fensters	204
11.2	Vorbereitungen für SAP GUI-Scripting	207
11.2.1	SAP GUI Scripting aktivieren	207
11.2.2	SAP GUI-Skripte mit dem Makrorecorder aufzeichnen	208
11.2.3	Anmeldung am System	209
11.3	SAP GUI-Skripte ausführen	209
11.3.1	Skript im Makrorecorder starten	210
11.3.2	Skript eigenständig im Windows Script Host starten.	211
11.4	Demo-Anwendung „Datenpflege“	213
11.4.1	SAP GUI-Skript aus der Wirtsanwendung Excel starten.	213
11.4.2	Daten in Excel bereitstellen	215
11.4.3	Code-Teile aufzeichnen	216
11.4.4	Code bearbeiten und zusammenstellen	218
11.4.5	Fehlerbehandlung.	219
11.4.5.1	Fehler „Falsches Material“	219
11.4.5.2	Fehler „Falsche Organisationseinheit“	220
11.4.5.3	Fehler „Falscher Wert“	221
11.4.6	Zusammenfassung	223
11.5	Tipps für das weitere Vorgehen	223
11.5.1	Scrollen	223
11.5.2	Datentabellen herunterladen	223
11.5.3	GUI-Elemente mit dem Scripting Tracker identifizieren.	223
11.6	Fazit und Ausblick	224
Literatur.	224

SAP GUI Scripting bietet schon seit etwa 2002 eine leicht zugängliche Möglichkeit, Abläufe in der ERP-Software von SAP zu automatisieren und mit Office-Anwendungen zu integrieren. Heute greifen auch Tools für Robotic Process Automation wie Power Automate, AutomationAnywhere und UI Path auf SAP GUI Scripting zurück, um SAP-Automatisierungen zu realisieren [1]–[3].

SAP GUI-Skripte finden viele Einsatzmöglichkeiten. Sie beschleunigen Dateneingaben, die sich oft in ähnlicher Form wiederholen, zum Beispiel beim Anlegen oder Rückmelden von Aufträgen oder beim Einpflegen umfangreicher Stammdaten wie etwa Stücklisten und Arbeitspläne. Die einzugebenden Daten stammen dabei aus Vorgänger-Prozessschritten oder werden mit Tools wie Excel vorbereitet und anschließend automatisch durch ein Makro in Eingabefelder der SAP GUI übertragen. Auch wenn turnusmäßig Daten aus SAP analysiert und zu Berichten aufbereitet werden, leisten SAP GUI-Skripte gute Dienste.

Nach einer Einführung in das Objektmodell der SAP GUI beschreibt dieses Kapitel, wie man SAP GUI Scripting aktiviert und wie man bei der Skriptentwicklung vorgehen kann. Wie ein menschlicher User, der mit einem SAP ERP-System arbeitet, muss auch ein SAP GUI-Skript am System angemeldet sein. Dafür gibt es mehrere Optionen. Das Kapitel diskutiert diese und beschreibt außerdem verschiedene Möglichkeiten, um ein SAP GUI-Skript zu starten. In diesem Zusammenhang wird gezeigt, wie sich auch andere VBA-Skripte mit dem Windows Script Host außerhalb einer Wirtsanwendung ausführen lassen. Das Kapitel präsentiert die Demo-Anwendung „Datenpflege“, die Materialstammdaten aktualisiert, und gibt Tipps und Hinweise auf weitere interessante Anwendungsmöglichkeiten.

11.1 Objektmodell der SAP GUI

SAP GUI Scripting automatisiert, wie auch der Name ausdrückt, nicht die ERP-Software direkt, sondern die SAP GUI. Die SAP GUI ist eine eigenständige Anwendung, die lokal auf Arbeitsplatzrechnern installiert ist und als Client Daten mit der eigentlichen ERP-Software austauscht, die zentral auf Anwendungsservern läuft. SAP GUI Scripting funktioniert daher für verschiedene Versionen der SAP ERP-Software weitgehend gleich, von SAP R/3 bis SAP S4/HANA.

Ein SAP GUI-Skript imitiert Benutzereingaben in die SAP GUI. Dieser Ansatz bringt Vorteile, aber auch Besonderheiten für die Skripterstellung mit sich. Ein Vorteil ist, dass SAP GUI-Skripte nicht mit dem komplexen Objektmodell der ERP-Software operieren, sondern mit dem wesentlich einfacheren Objektmodell der SAP GUI. Im Objektmodell der SAP GUI genügt eine überschaubare Anzahl von GUI-Elementen, um viele verschiedene Datenansichten und Eingabemasken für das ERP-System darzustellen. Ein weiterer Vorteil ist, dass sich ein SAP GUI-Skript nicht mit dem Berechtigungssystem in SAP auseinanderzusetzen braucht, denn ein Skript läuft immer mit den Anmelddaten eines Nutzerkontos und erbt dessen Zugriffsrechte und Einstellungen. Der Ansatz

bringt aber auch mit sich, dass ein SAP GUI-Skript alle Aktionen nachvollziehen muss, die auch ein menschlicher Benutzer ausführen würde, um Daten abzurufen oder ändern. Dazu gehört beispielsweise auch das Scrollen in GUI-Fenstern und in Tabellen mit vielen Zeilen.

11.1.1 Objekte des SAP GUI-Objektmodells

Das Objektmodell der SAP GUI bildet nicht die Elemente und Funktionen der Anwendung SAP ERP ab, sondern die Elemente der SAP GUI. Abb. 11.1 zeigt einen kleinen Ausschnitt aus diesem Objektmodell. Eine vollständige Dokumentation ist auf dem SAP Help Portal zu finden [4].

Die Toplevel-Objekte des Objektmodells sind GuiApplication, GuiConnection und GuiSession. Die GuiApplication ist die Automatisierungs komponente der SAP GUI. Sie verwaltet Objekte des Typs GuiConnection. Eine GuiConnection ist eine Verbindung der SAP GUI zu einem SAP Anwendungsserver.

Eine GuiConnection verwaltet Objekte des Typs GuiSession. Die GuiSession definiert den Benutzerkontext für alle Aktionen mit dem System, also seine Berechtigungen, Voreinstellungen etc. Sie besitzt ein Feld Info, das ein Objekt des Typs GuiSessionInfo enthält. Die GuiSessionInfo beschreibt die GuiSession. Wenn in einer GuiSession ein Benutzer am System SAP angemeldet ist, speichert die GuiSessionInfo typischerweise unter anderem den Namen des verbundenen SAP-Anwendungsservers, den SAP-Anmeldenamen des angemeldeten Benutzers, den SAP-Mandanten, auf den die Session zugreift, und die SAP-Transaktion, welche die Session aktuell ausführt. Zu einer GuiSession gehört ein GuiMainWindow, also ein Fenster der SAP GUI, in dem ein Benutzer oder ein Skript mit dem System interagiert. Optional können in

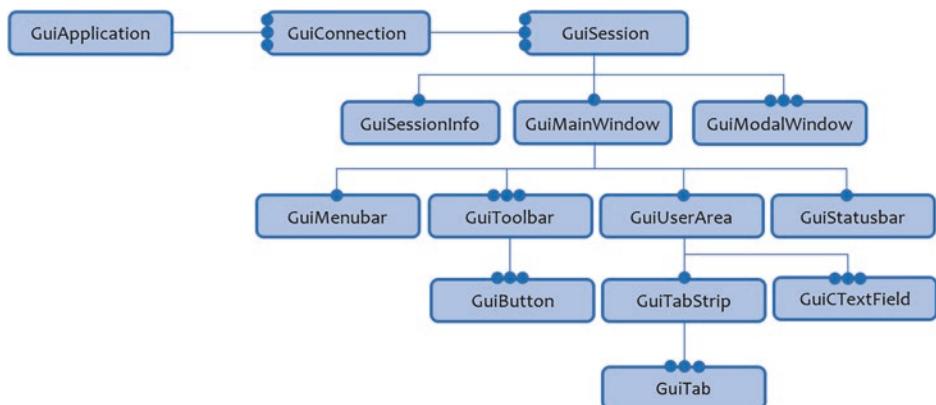


Abb. 11.1 Ausschnitt aus dem Objektmodell der SAP GUI

einer GuiSession ein oder mehrere weitere Dialoge offen sein. Diese sind dann Objekte des Typs GuiModalWindow. Ein GuiMainWindow kann unterschiedliche GUI-Elemente enthalten wie GuiMenuBar, GuiToolBar, GuiCTextField und weitere.

11.1.2 Aufbau eines SAP GUI-Fensters

Abb. 11.2 zeigt ein Beispiel für ein GuiMainWindow. Dieses GUI-Fenster besitzt einen ersten GuiToolbar, dessen GuiButtons ③, ⑦, ⑧ und ⑨ sich über das Fenster verteilen, einen zweiten GuiToolbar ①, einen GuiMenubar ② und einen GuiStatusbar ⑥, der jedoch im abgebildeten Screenshot gerade keine Meldung zeigt. Die größte Fläche des Fensters nimmt die GuiUserArea ④ ein. Diese enthält hier einen

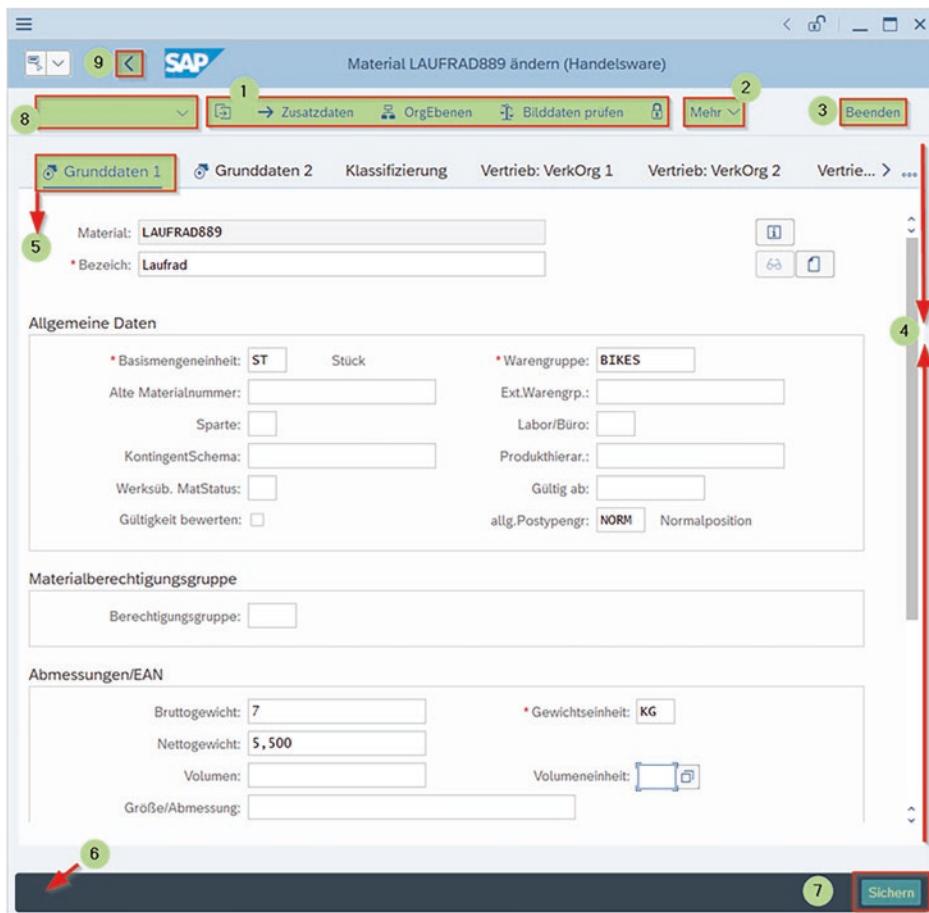


Abb. 11.2 Beispiel für ein SAP GUI-Fenster: MM02

GuiTabStrip (ohne eigene Nummer) mit mehreren GuiTabs. Der erste GuiTab „Grunddaten 1“ ⑧ ist selektiert. Auf diesem Tab sind unter anderem mehrere Elemente des Typs GuiCTextField sichtbar.

Aus Sicht eines Skripts ist ein SAP GUI-Fenster baumartig oder hierarchisch aufgebaut. Es besteht aus einfachen GUI-Elementtypen wie GuiCTextField, GuiTextField und GuiButton, die Werte anzeigen oder Aktionen auslösen, und aus sogenannten Container-Objekttypen wie etwa GuiMainWindow, GuiUserArea, GuiTabStrip und GuiMenubar, die weitere untergeordnete Container-Objekte und einfache GUI-Elemente enthalten.

Container-Objekte verwalten die ihnen direkt untergeordneten Elemente in einem Feld Children. Zum Beispiel besitzt ein GuiTabStrip-Objekt als Children Objekte des Typs GuiTab. Ein GuiMenubar besitzt als Children GuiButtons und optional das GuiOkCodeField (in Abb. 11.2 mit ⑧ markiert). Das Children-Feld verweist auf eine Collection, die sich in einer Schleife durchlaufen lässt. In einigen Fällen enthält sie jedoch nicht alle vorhandenen GUI-Elemente, sondern nur die aktuell auf dem Bildschirm sichtbaren.

Insgesamt ergibt dies eine hierarchische Baumstruktur, in der sich die einzelnen GUI-Elemente über Pfade ansteuern lassen. Der Zugriff auf einzelne GUI-Elemente über Pfade ist eine zentrale Aktivität in einem SAP GUI-Skript, denn in den GUI-Elementen werden Daten eingegeben oder gelesen und Aktionen gestartet.

Das Objektmodell kennt mehrere Methoden, um ein GUI-Element zu identifizieren. Häufig wird die Methode findById verwendet, die im Session-Objekt und in allen Container-Objekten zur Verfügung steht. Beispiele für die Ansteuerung von GUI-Elementen zeigen die folgenden Befehle:

```
(1) session.findById("wnd[0]/usr/tabsTABSPR1/tabpSP04").select
(2) session.findById("wnd[0]/usr/tabsTABSPR1/tabpSP01").select
(3) session.findById("wnd[0]/usr/tabsTABSPR1/tabpSP01/ssubTABFRA1:_SAPLMGMM:2004/subSUB1:SAPLMDG1:1002/txtMAKT-MAKTX").text = "Lauftrad"
(4) session.findById("wnd[0]/tbar[0]/btn[11]").press
(5) session.findById("wnd[0]/tbar[0]/okcd").text = "/nMM02"
(6) session.findById("wnd[0]").sendVKey 0
```

Die Befehle verwenden die findById-Methode des GuiSession-Objekts, das hier in einer Variablen session gespeichert ist. Sie beziehen sich auf das GUI-Fenster in Abb. 11.2. Die Baumstruktur dieses SAP GUI-Fensters ist auch in Abb. 11.3 zu sehen, dargestellt im Tool Scripting Tracker, das in Abschn. 11.5.3 noch genauer beschrieben wird. Die Parameter von findById sind Pfade durch die Struktur der GUI-Elemente. Diese werden nun im Detail betrachtet.

Der Ausdruck wnd[0] bezeichnet das erste Fenster der Session, also das GuiMainWindow. Usr ist die GuiUserArea des Hauptfensters. Der Ausdruck tabsTABSPR1



Abb. 11.3 Aufbau eines SAP GUI-Fensters, angezeigt im Scripting Tracker [5]

identifiziert das Element GuiTabStrip in der GuiUserArea usr. Bei tabpSP01 und tabpSP04 handelt es sich um GuiTab-Objekte; hier sind dies die Reiter „Grunddaten 1“ beziehungsweise „Vertrieb: VerkOrg 1“. Die ersten beiden Befehle wählen diese GuiTab-Objekte mit der Methode select an.

Der Reiter tabpSP01 mit den „Grunddaten 1“ enthält den GuiScrollViewContainer ssubTABFRA1 und dieser wiederum den GuiSimpleContainer subSUB1. Diese beiden Container sind im GUI-Fenster in Abb. 11.2 nicht direkt erkennbar. Im GuiSimpleContainer subSUB1 liegt unter anderem das GuiTextField MAKTX-MAKTX. Der dritte Befehl des Code-Schnipsels weist diesem GuiTextField den Wert „Laufrad“ zu.

Der vierte Befehl des Code-Schnipsels betätigt den GuiButton mit dem Index 11, der zur ersten GuiToolbar tbar[0] des GuiMainWindow wnd[0] gehört. Dabei handelt es sich um die Schaltfläche „Sichern“, die in Abb. 11.2 mit ⑦ markiert ist.

Der fünfte Befehl wirkt ebenfalls auf ein Element in der GuiToolbar tbar[0], nämlich das GuiOkCodeField okcd, das in Abb. 11.2 mit ⑧ markiert ist. Der Befehl trägt dort das Kommando „/nMM02“ ein, das die Transaktion **MM02** in einem weiteren GUI-Fenster öffnet. Der letzte der Beispielbefehle sendet die Eingabetaste an das GuiMainWindow wnd[0].

Das Hauptfenster einer GuiSession ist immer wnd[0]. Wenn die Session weitere GUI-Fenster öffnet, handelt es sich dabei um modale Dialoge, zum Beispiel Suchfenster, Untermenüs oder Auswahlfenster. Diese werden durchnummieriert als wnd[1], wnd[2] und so weiter.

GUI-Objekte lassen sich Variablen zuweisen, wie beispielsweise die GuiStatusbar im folgenden Code-Schnipsel. In Abb. 11.2 ist die GuiStatusbar mit ⑨ markiert.

```
Dim statuszeile as GuiStatusbar
Set statuszeile = SAPsession.findById("wnd[0]/sbar")
If statuszeile.MessageType = "E" Then
    MsgBox("Ein Fehler ist aufgetreten: " & statuszeile.Text)
End if
```

Wie kann man die Pfade der GUI-Elemente ermitteln? Die SAP GUI bringt einen Makrorecorder mit, der Benutzeraktionen in einem SAP GUI-Fenster als Skript aufzeichnet. Diese Skripte verwenden die `findById`-Methode des `Session`-Objekts wie im oben erläuterten Code-Schnipsel und zeigen somit die Pfade der betätigten GUI-Elemente an. Auch Tools wie der Scripting Tracker helfen dabei, GUI-Elemente zu identifizieren. Eine Ausgabe des Scripting Tracker ist in Abb. 11.3 sehen. Der Scripting Tracker wird in Abschn. 11.5.3 nochmals kurz angesprochen.

11.2 Vorbereitungen für SAP GUI-Scripting

Um SAP GUI-Skripte entwickeln und ausführen zu können, sind einige Voraussetzungen zu schaffen. Diese werden im Folgenden genannt. Auch wird der SAP GUI-Makrorecorder vorgestellt, der ein wichtiges Hilfsmittel für die Skriptentwicklung ist.

11.2.1 SAP GUI Scripting aktivieren

Um SAP GUI Scripting nutzen zu können, sind Einstellungen an zwei Stellen erforderlich:

- In der SAP GUI muss die Skriptunterstützung aktiviert sein.
- Im SAP-Anwendungssystem muss der Parameter `sapgui/user_scripting` auf `True` stehen.

Abb. 11.4 zeigt das Dialogfenster mit den Einstellungen zur Skriptunterstützung in der SAP GUI. Es zeigt an, ob SAP GUI Scripting auf dem jeweiligen Arbeitsplatzrechner installiert ist. Ist dies der Fall, lässt sich die Skriptunterstützung hier aktivieren und konfigurieren. Auch eine Hilfe zu den Einstellungsmöglichkeiten ist hier verlinkt. Der Pfad, über den dieses Dialogfenster zu erreichen ist, hängt ab von der Version der SAP GUI und vom gewählten Theme. In aktuellen GUI-Versionen und im Theme Belize liegt es unter **Mehr > SAP GUI – Einstellungen+Aktionen > Optionen > Barrierefreiheit & Scripting > Skriptunterstützung**.

Den Parameter `sapgui/user_scripting` findet und setzt man in der SAP-Transaktion **RZ11** wie folgt:

- Die Transaktion **RZ11** öffnen.
- Im Feld Parametername den Wert `sapgui/user_scripting` eingeben.

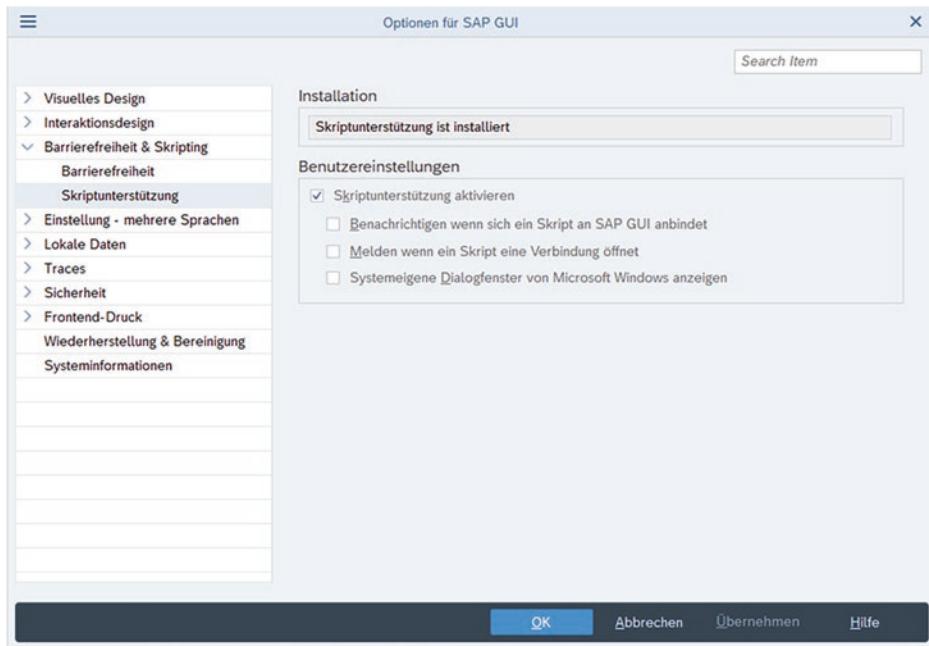


Abb. 11.4 Aktivierung und Einstellmöglichkeiten für SAP GUI Scripting

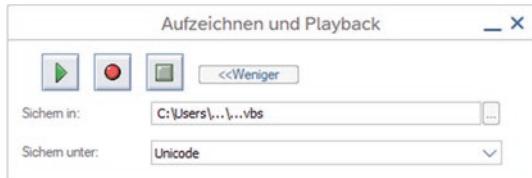
- Die Schaltfläche „Anzeige“ anklicken.
- Es erscheint eine etwas unübersichtliche Tabelle, die unter anderem eine Zeile „Aktueller Wert“ enthält. Hier muss der Eintrag TRUE stehen.
- Wenn dies nicht der Fall ist, klickt man auf die Schaltfläche „Wert ändern“.
- Im Feld „Neuer Wert“ ersetzt man den Eintrag FALSE durch TRUE und speichert.

Parameteränderungen in der Transaktion **RZ11** werden sofort wirksam, gehen aber bei einem Neustart des Systems auf dem Anwendungsserver verloren und müssen dann erneut vorgenommen werden.

11.2.2 SAP GUI-Skripte mit dem Makrorecorder aufzeichnen

Die SAP GUI enthält einen Makrorecorder, der die Aktionen eines Benutzers in der GUI aufzeichnet und aufgezeichnete Skripte abspielt. Der Makrorecorder lässt sich öffnen über **Mehr > SAP GUI – Einstellungen+Aktionen > Skript-Aufzeichnung und -Playback** oder ähnlich, je nach GUI-Version und Theme. Er generiert Skripte in der Sprache VBScript, einer einfachen Version von Visual Basic, die sich ohne wesentliche Änderungen in VBA-Skripte integrieren lässt. VBScript-Dateien sind Textdateien und haben

Abb. 11.5 Der Makrorecorder der SAP GUI



üblicherweise die Endung „.vbs“. Sie lassen sich mit beliebigen Texteditoren bearbeiten und auch in die VBA-Entwicklungsumgebung einer Office-Anwendung importieren.

Wie Abb. 11.5 zeigt, ist der Makrorecorder übersichtlich und leicht zu bedienen. Die Schaltfläche mit dem Kreis startet eine Skriptaufzeichnung. Die Schaltfläche mit dem Viereck stoppt eine laufende Aufzeichnung. Der Makrorecorder speichert ein aufgezeichnetes Skript in eine Textdatei mit der Dateiendung „.vbs“. Die Schaltfläche mit dem Dreieck erlaubt, vorhandene Skripte zu laden und auszuführen.

SAP GUI-Skriptentwicklung beginnt meist damit, Abläufe ganz oder in Teilen mit dem Makrorecorder aufzuzeichnen, die entstandenen Skripte anzupassen, zusammenzufügen und mit weiterem VBA-Code zu integrieren.

11.2.3 Anmeldung am System

Ein SAP GUI-Skript imitiert einen menschlichen Benutzer, der am System angemeldet ist. Die Einstellungen und Zugriffsberechtigungen dieses Benutzers gelten dann auch für das Skript. Es wäre aus technischer Sicht kein Problem, ein Skript zu erstellen, das sich mit einem gültigen Benutzernamen und Passwort selbst in SAP anmeldet. Dazu müssten jedoch diese Logindaten unverschlüsselt als Klartext im Skript hinterlegt sein und davon ist aus Sicherheitsgründen abzuraten [6]. Stattdessen greifen SAP GUI-Skripte üblicherweise auf eine SAP-Session zu, die ein Benutzer manuell gestartet und in der er sich bereits manuell angemeldet hat. Auch die Code-Beispiele und die Demo-Anwendung dieses Kapitels setzen voraus, dass bei ihrem Start bereits ein Benutzer am System angemeldet ist.

11.3 SAP GUI-Skripte ausführen

Ein SAP GUI-Skript lässt sich auf mehrere Arten ausführen. So kann man es wie oben beschrieben aus dem SAP GUI-Makrorecorder heraus starten. SAP GUI-Skripte laufen auch eigenständig, das heißt, ohne Wirtsanwendung oder Makrorecorder, mit dem Windows Script Host. Drittens kann man ein SAP GUI-Skript in eine andere Wirtsanwendung einbetten und mit dort entwickeltem VBA-Code integrieren. Die folgenden beiden Unterabschnitte zeigen und diskutieren Befehlssequenzen, um Skripte, die im

SAP GUI-Makrorecorder oder im Windows Script Host ausgeführt werden sollen, mit einer SAP-Session zu verbinden. Der Abschn. 11.4 zeigt dann im Kontext der Demo-Anwendung „Datenpflege“ eine entsprechende Befehlssequenz für ein in der Wortsanwendung Excel laufendes Skript.

11.3.1 Skript im Makrorecorder starten

Skripte, die der Makrorecorder der SAP GUI generiert, beginnen mit der unten gezeigten Befehlssequenz. Ihr Zweck ist es, das Skript mit einer SAP GUI-Session zu verbinden. Darauf folgt dann der Code, der die eigentlichen Aktionen ausführt. Die Befehlssequenz weist einige Besonderheiten auf und muss modifiziert werden für ein SAP GUI-Skript, das nicht aus dem Makrorecorder gestartet werden soll. Daher wird sie nun näher betrachtet.

```
If Not IsObject(application) Then
    Set SapGuiAuto = GetObject("SAPGUI")
    Set application = SapGuiAuto.GetScriptingEngine
End If
If Not IsObject(connection) Then
    Set connection = application.Children(0)
End If
If Not IsObject(session) Then
    Set session = connection.Children(0)
End If
If IsObject(WScript) Then
    WScript.ConnectObject session,     "on"
    WScript.ConnectObject application, "on"
End If
```

Die Befehle greifen nacheinander auf die SAP GUI-Automatisierungskomponente, die erste vorhandene Connection und deren erste Session zu und binden sie an entsprechende Variablen. Vorher prüft der Code jeweils, ob die betreffende Variable schon belegt ist. Wenn ja, behält er die vorhandene Belegung bei. Diese Prüfungen sind relevant, wenn ein Skript aus dem SAP GUI-Makrorecorder heraus gestartet wird. Wenn der Makrorecorder über eine Schaltfläche im Menü eines GUI-Fensters geöffnet wurde, ist die Session-Variablen des Skripts mit der Session dieses GUI-Fensters vorbelegt, sodass das Skript auf dieses GUI-Fenster wirkt. Die vorhandene Session soll dann nicht mit einer möglicherweise vorhandenen weiteren Session überschrieben werden, die mit einem anderen GUI-Fenster verbunden ist.

Das WScript-Objekt, das in den letzten Befehlen dieses Code-Schnipsels vorkommt, repräsentiert eine Instanz des Windows Script Host (mehr dazu im nächsten Abschnitt). Immer wenn das SAP GUI-Makro im Windows Script Host läuft, exis-

tiert ein WScript-Objekt. Die Befehle bewirken dann, dass der Windows Script Host vom Session- und vom Application-Objekt generierte Ereignisse empfangen und verarbeiten kann [7]. Wenn jedoch das SAP GUI-Makro nicht im Windows Script Host läuft, existiert das WScript-Objekt nicht und die Befehle würden einen Fehler auslösen. Um dies zu verhindern, prüft der Code zuerst, ob das WScript-Objekt vorhanden ist.

Dieser Code-Schnipsel läuft erfolgreich, wenn die SAP GUI offen und mindestens eine mit einem SAP-Anwendungssystem verbundene Session vorhanden ist. Andernfalls endet der Code-Schnipsel mit einem Fehler. Man kann Skripte, die mit dieser Befehlssequenz beginnen, im SAP GUI-Makrorecorder oder im Windows Script Host ausführen. Diese zweite Möglichkeit betrachtet der folgende Unterabschnitt.

11.3.2 Skript eigenständig im Windows Script Host starten

Der Windows Script Host ist eine Laufzeitumgebung für Skripte, die das Betriebssystem Windows mitbringt. Er akzeptiert Skripte in diversen Formaten, neben VBScript zum Beispiel auch JavaScript. Der Windows Script Host ist standardmäßig auf Windows-Rechnern aktiviert und als Standardanwendung für Dateien mit der Endung „.vbs“ hinterlegt. Ein Doppelklick auf eine Skript-Datei mit der Endung „.vbs“ genügt, um das Skript zu starten. Dies gilt auch für die Skriptdateien, die der SAP GUI-Makrorecorder erzeugt.

VBScript kann wie VBA Objektmodelle einbinden und nutzen. Die vom SAP GUI-Makrorecorder erzeugte Befehlssequenz bindet das Objektmodell der SAP GUI per Late Binding ein. Der folgende Code-Schnipsel zeigt, wie auch das Excel-Objektmodell in ein VBScript eingebunden werden kann.

Um diesen Code-Schnipsel auszuführen, kopiert man ihn in eine Textdatei und speichert diese mit der Endung „.vbs“ ab, zum Beispiel als „initexcel.vbs“. Wenn Windows standardmäßig eingerichtet ist, startet ein Doppelklick auf den Dateinamen dieses Skript. Es prüft zunächst, ob aktuell eine Excel-Instanz läuft. Wenn nicht, startet es eine Instanz von Excel. Wenn keine Arbeitsmappe offen ist, erzeugt es eine neue Arbeitsmappe. Dann schreibt es in die erste Zelle des ersten Arbeitsblatts der ersten geöffneten Arbeitsmappe den Text „Hello World“. Dieses Skript kann also eine VBA-Automatisierung ausführen, ohne dass Benutzer zuvor eine Wirtsanwendung öffnen müssen.

```
On Error Resume Next
Set exApp = GetObject( , "Excel.Application")

If Err.Number <> 0 Then
    Set exApp = CreateObject("Excel.Application")
End If
On Error GoTo 0
exApp.Visible = True
```

```
If exApp.Workbooks.Count < 1 Then
    exApp.Workbooks.Add
End If
exApp.Workbooks(1).Worksheets(1).Cells(1, 1) = "Hello World"
```

Dieses VB-Skript bearbeitet das erste Arbeitsblatt der ersten Arbeitsmappe, die es findet. Ähnlich verhält sich die Befehlssequenz, die der SAP GUI-Makrorecorder generiert, wenn sie außerhalb des Makrorecorders gestartet wird: Sie greift auf die erste Session der ersten Connection zu, die sie findet. Dieses Verhalten ist offensichtlich fehlerträchtig, da die SAP GUI mehrere Verbindungen und Sessions gleichzeitig offenhalten kann, zum Beispiel Verbindungen zu einem Entwicklungssystem und einem Produktivsystem. Ein SAP GUI-Skript kann damit leicht versehentlich im falschen System Operationen ausführen.

Um eine Fehlanwendung zu verhindern, sind verschiedene Optionen gebräuchlich. Eine Option besteht darin, im Skript die Anzahl offener Connections und Sessions zu prüfen und die Ausführung abzubrechen, wenn mehrere Connections oder Sessions offen sind. Ein anderer Ansatz hinterlegt Zielsystem, Mandant und vielleicht auch den SAP-Nutzer im Skript und lässt das Skript prüfen, ob die gefundene erste Session mit dem hinterlegten Benutzer mit dem hinterlegten Zielsystem verbunden ist. Wenn nicht, bricht das Skript ab. Als Variante dieses Ansatzes durchsucht das Skript mit einer Schleife alle offenen Connections und Sessions, bis es eine Session findet, die mit dem angegebenen Zielsystem und Mandanten verbunden ist.

Die folgende Befehlssequenz „initscript.vbs“ implementiert eine weitere Lösung: Sie greift ebenfalls auf die erste Session der ersten Verbindung zu, zeigt dann aber Infos zu dieser Session an und arbeitet nur damit weiter, wenn der User dies explizit bestätigt. Infos zur jeweiligen Session bezieht sie aus dem GuiSessionInfo-Objekt.

```
On Error Resume Next
Set SapGuiAuto = GetObject("SAPGUI")
Set Application = SapGuiAuto.GetScriptingEngine
Set connection = Application.Children(0)
Set session = connection.Children(0)
Set SessionInfo = session.info

If Err.Number <> 0 Then
    MsgBox "Keine SAP Session gefunden. Ende.", , "SAP GUI Skript"
    WScript.Quit
End If
On Error GoTo 0
WScript.ConnectObject session, "on"
WScript.ConnectObject Application, "on"

With SessionInfo
    msgResult = MsgBox("System: " & .SystemName & _
```

```
" Mandant: " & .Client & _  
" User: " & .User & " Tcode: " & .Transaction & vbLf & _  
"Starten?", vbYesNo, "SAP GUI Skript")  
End With  
  
If msgResult <> vbYes Then  
    MsgBox "Gestoppt.", , "SAP GUI Skript"  
    WScript.Quit  
End If  
MsgBox "Skript beginnt...", , "SAP GUI Skript"
```

Diese Befehlssequenz unterscheidet sich von der Version, die der Makrorecorder generiert. Sie berücksichtigt, dass beim eigenständigen Start des Skripts die SAP GUI nicht unbedingt läuft. Das Objekt `WScript` dagegen ist sicher vorhanden, wenn das Script im Windows Script Host ausgeführt wird. Deshalb verzichtet die Befehlssequenz auf die Abfrage `IsObject (WScript)`, die der SAP GUI-Makrorecorder einfügen würde.

Diese Befehlssequenz eignet sich, um Skripte, die im Windows Script Host ausgeführt werden, mit einem SAP ERP-System zu verbinden. Eine Möglichkeit zur Anmeldung von Skripten, die in einer Wirtsanwendung wie Excel laufen, zeigt die Demo-Anwendung im folgenden Abschnitt.

11.4 Demo-Anwendung „Datenpflege“

SAP GUI-Skripte sind ein wirksames Hilfsmittel, wenn bei der Datenpflege viele Daten zu aktualisieren sind. Die Demo-Anwendung „Datenpflege“ überträgt Daten von einem Excel-Arbeitsblatt in die Materialstammdaten in einem SAP ERP-System. Während die eigentliche Datenaktualisierung sehr einfach zu implementieren ist, erfordert die Behandlung und Protokollierung etwaiger Fehler mehr Aufwand.

Die Demo-Anwendung läuft in der Wirtsanwendung Excel. Im Excel-VBA-Projekt muss dafür ein Verweis auf die SAP GUI Scripting API gesetzt werden. Falls diese in der Liste verfügbarer Verweise nicht auftaucht, lädt man sie mit der Funktion „Durchsuchen“ vom Dateiverzeichnissystem wie in Abb. 11.6. Die Abbildung zeigt die betreffende Datei und den Pfad, an dem sie typischerweise zu finden ist.

11.4.1 SAP GUI-Skript aus der Wirtsanwendung Excel starten

Der folgende Code-Schnipsel zeigt eine Möglichkeit, um ein SAP GUI-Skript aus der Wirtsanwendung Excel starten und mit dem SAP ERP-System zu verbinden. Sie unterscheidet sich in einigen Punkten von den Versionen, die in Abschn. 11.3 diskutiert wurden: Wichtig ist zunächst, die Variable `Application` umzubenennen, etwa wie hier in `sapApp`, da `Application` in Excel-VBA für die `Application` des Excel-Objekts steht.

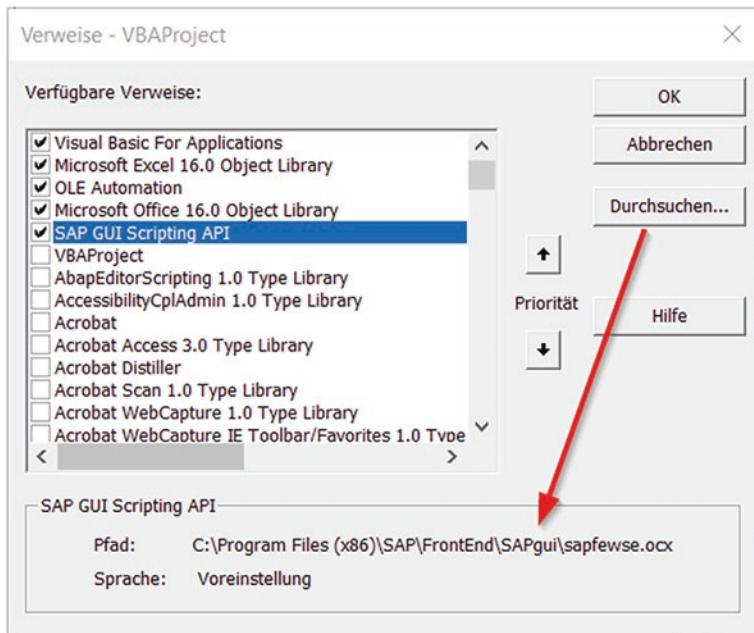


Abb. 11.6 Einbinden der SAP GUI Scripting API

modells steht. Weiterhin sind die Befehle, welche die SAP GUI-Ereignisse im Windows Script Host registrieren, überflüssig und können entfernt werden, weil das Skript in Excel und nicht im Windows Script Host läuft.

Die Demo-Anwendung „Datenpflege“ verarbeitet keine Ereignisse der SAP GUI (zu Ereignissen siehe auch Abschn. 13.5.1). Einen Weg, wie man bei Bedarf SAP GUI-Ereignisse in Excel-VBA verfügbar machen könnte, findet man in einem Forumsbeitrag von S. Azmi [8].

```

Sub Starten()
Dim SapGuiAuto
Dim sapApp As SAPFEWSELib.GuiApplication
Dim connection As SAPFEWSELib.GuiConnection
Dim session As SAPFEWSELib.GuiSession

On Error GoTo START_FEHLER
Set SapGuiAuto = GetObject("SAPGUI")
Set sapApp = SapGuiAuto.GetScriptingEngine
Set connection = sapApp.Children(0)
Set session = connection.Children(0)
On Error GoTo 0

```

```

If (sapApp.Children.Count > 1) Or connection.Children.Count > 1 Then
    MsgBox "SAP-Anmeldung nicht eindeutig. " _
        & "Bitte nur einmal anmelden!", , "SAP GUI Skript"
    Exit Sub
End If
If (session.info.User = vbNullString) Then
    MsgBox "Kein SAP-User angemeldet. " _
        & "Bitte am System anmelden!", , "SAP GUI Skript"
    Exit Sub
End If
Bearbeiten session ' hier passiert etwas Nützliches
Exit Sub

START_FEHLER:
MsgBox "Keine SAP-Session gefunden. " _
    & "Bitte SAP starten und anmelden!", , "SAP GUI Skript"
End Sub

```

Die Sub Starten arbeitet mit einer bestehenden SAP-Verbindung. Sie sucht eine GuiSession, in der ein Benutzer am ERP-System angemeldet ist, und bricht ab, wenn sie keine solche GuiSession findet. Sie bricht ebenfalls ab, wenn sie mehr als eine Verbindung entdeckt, und sichert sich so dagegen ab, mit einer falschen GuiSession zu arbeiten.

11.4.2 Daten in Excel bereitstellen

Die Demo-Anwendung „Datenpflege“ aktualisiert oder vervollständigt Materialstamm-daten, indem sie Werte in der Sicht „Grunddaten 1“ der Transaktion **MM02** einpflegt, wie in Abb. 11.2. Materialien und Werte sind in einem Excel-Arbeitsblatt wie in Abb. 11.7 vorbereitet. Jede Zeile entspricht einem Datensatz. Die ersten vier Spalten „Material“,

	A	B	C	D	E	F	G	H	I
1	Material	Werk	VerkaufsOrg	Vertriebsweg	Volumen	VollEinheit	Groesse	Status	StatusText
2	Laufrad889	HD00	DS00	WH	0,44	M4	2x3x3,2	E	Maßeinheit M4 ist in Sprache DE nicht angelegt
3	Laufrad888	HD00	DS00	WH	0,44	M3	6x3x3,2	S	Das Material LAUFRAD888 wird geändert
4	Laufrad889	HD00	DS00	WH	0,44	M3	2x3x3,2	S	Es wurden keine Änderungen durchgeführt
5	Laufrad890	HD00	DS00	WH	0,44	M3	2x3x3,2	E	Das Material LAUFRAD890 ist nicht vorhanden oder nicht aktiviert
6	Laufrad889	HD00	DS01	WH	1,44	M4	2x3x3,2	E	Eintrag D501 nicht vorhanden in TVKO
7	Laufrad000	HD00	DS00	WH	0,44	M3	2x3x3,2	E	Das Material existiert nicht in VerkOrg. DS00
8									
	2022_11_11								
	Bereit								

Abb. 11.7 Beispieldaten zur Demo-Anwendung „Datenpflege“

„Werk“, „VerkaufsOrg“ und „Vertriebsweg“ identifizieren den Datensatz. Wenn man die Demo-Anwendung ausführt, muss im verbundenen SAP-System ein Datensatz zu diesen Werten existieren. Die Spalten „Volumen“, „VolEinheit“ und „Groesse“ enthalten die einzupflegenden Werte. Zwar ist die Angabe von „Werk“, „VerkaufsOrg“ und „Vertriebsweg“ eigentlich nicht nötig, wenn nur die Grunddaten eines Materials bearbeitet werden. Die Demo-Anwendung wählt jedoch eine weitere Sicht aus, um den Umgang mit den Dialogen für die Auswahl der Sichten und der Organisationsebenen sowie einige Fehlerbehandlungen zu zeigen. In den Spalten „Status“ und „StatusText“ protokolliert das Demo-Skript, ob es den Datensatz erfolgreich bearbeiten konnte. Status „S“ bedeutet „Success“, Status „E“ signalisiert einen Fehler.

11.4.3 Code-Teile aufzeichnen

Um eine Anwendung mit SAP GUI-Skripten zu entwickeln, führt man einen Bearbeitungsvorgang manuell durch und zeichnet ihn mit dem Makrorecorder auf. Danach bearbeitet man den aufgezeichneten Code und fügt ihn mit manuell erstelltem VBA-Code zusammen.

Die folgende Befehlssequenz hat der Makrorecorder bei der Auswahl eines Materials aufgezeichnet. Sie ist weitgehend unbearbeitet. Lediglich das Session-Objekt ist umbenannt in ss, damit die Codezeilen kürzer werden. Kommentare erklären, was diese Codezeilen bewirken. Sie beziehen sich auf die jeweils vorangehenden Codezeilen. Einige Codezeilen, die überflüssig sind und entfernt werden können, sind mit einem Doppelkreuz # markiert:

```
ss.findById("wnd[0]").resizeWorkingPane 115,28,false
' bringt das GUI-Fenster auf eine fixe Größe. Alternativ:
' ss.findById("wnd[0]").Maximize
ss.findById("wnd[0]/tbar[0]/okcd").Text = "/nMM02"
' trägt den Transaktionscode MM02 ein
ss.findById("wnd[0]").sendVKey 0
' Eingabetaste, öffnet die Transaktion MM02
ss.findById("wnd[0]/usr/ctxtRMMG1-MATNR").Text = "LAUFRAD889"
ss.findById("wnd[0]").sendVKey 0
' wählt ein Material aus
ss.findById("wnd[1]/usr/tblSAPLMGMMTC_VIEW").GetAbsoluteRow(0) _
    .Selected = True
ss.findById("wnd[1]/usr/tblSAPLMGMMTC_VIEW").GetAbsoluteRow(3) _
    .Selected = True
' setzt im Sichtenauswahl-Dialog den Haken bei zwei Sichten
ss.findById("wnd[1]/usr/tblSAPLMGMMTC_VIEW/txtMSICHTAUSW-DYTXT[0,3]") _
    .setFocus
' # setzt den Focus auf die Beschriftung einer Sicht
ss.findById("wnd[1]/usr/tblSAPLMGMMTC_VIEW/txtMSICHTAUSW-DYTXT[0,3]"). _
```

```

caretPosition = 0
' # platziert den Cursor vor das erste Zeichen der Beschriftung
ss.findById("wnd[1]/tbar[0]/btn[0]").press
' betätigt den Button ok
ss.findById("wnd[1]/usr/ctxtRMMG1-WERKS").Text = "HD00"
ss.findById("wnd[1]/usr/ctxtRMMG1-VKORG").Text = "DS00"
ss.findById("wnd[1]/usr/ctxtRMMG1-VTWEG").Text = "WH"
' trägt im Organisationsebenen-Dialog Werte ein
ss.findById("wnd[1]/usr/ctxtRMMG1-VTWEG").setFocus
ss.findById("wnd[1]/usr/ctxtRMMG1-VTWEG").caretPosition = 2
' # setzt den Focus auf eine Beschriftung und positioniert den Cursor
ss.findById("wnd[1]/tbar[0]/btn[0]").press
' betätigt den Button ok

```

Damit ist nun ein Material ausgewählt und zum Bearbeiten in der Sicht „Grunddaten 1“ geöffnet. Die folgende Befehle befüllen darin drei Felder mit Werten:

```

ss.findById("wnd[0]/usr/tabsTABSPR1/tabpSP01/" & _
& "ssubTABFRA1:SAPLMGMM:2004/" & _
& "subSUB4:SAPLMGD1:2007/ctxtMARA-VOLUM").text = "0,70"
ss.findById("wnd[0]/usr/tabsTABSPR1/tabpSP01/" & _
& "ssubTABFRA1:SAPLMGMM:2004/" & _
& "subSUB4:SAPLMGD1:2007/ctxtMARA-VOLEH").text = "M3"
ss.findById("wnd[0]/usr/tabsTABSPR1/tabpSP01/" & _
& "ssubTABFRA1:SAPLMGMM:2004/" & _
& "subSUB4:SAPLMGD1:2007/ctxtMARA-GROES").text = _
= "0,68m x 0,34m x 0,8m"
' drei Datenfelder mit Werten füllen
ss.findById("wnd[0]/tbar[0]/btn[11]").press
' Button Sichern drücken

```

Wer übersichtlicheren Code möchte, kann diese Werte-Eingaben so umcodieren:

```

Dim mainw As SAPFEWSELib.GuiMainWindow
Set mainw = session.findById("wnd[0]")
mainw.FindByName("MARA-VOLUM", "GuiTextField").Text = "0,70"
mainw.FindByName("MARA-VOLEH", "GuiCTextField").Text = "M3"
mainw.FindByName("MARA-GROES", "GuiTextField").Text =
= "0,68m x 0,34m x 0,8m"

```

Hier ersetzt die Methode `FindByName` die Methode `findById`. Die Methode `FindByName` ist übersichtlicher, weil sie den Pfad nicht benötigt, dafür aber den Typ des GUI-Elements. Diesen kann man sich meist einfach erschließen, wobei auch die Dokumentation zum Objektmodell der SAP GUI hilft [4]. Zum Beispiel ist `txtMARA-GROES` ein `GuiTextField` und `ctxtMARA-VOLEH` ein `GuiCTextField`. In Abb. 11.2 sind

dies die Felder „Größe/Abmessung“ beziehungsweise „Volumeneinheit“. Wenn in einem GUI-Fenster die Kombination Feldname/Typ mehrfach vorkommt, funktioniert die Methode `FindByName` nicht.

11.4.4 Code bearbeiten und zusammenstellen

Es ist nun einfach, den bereinigten Code in eine Schleife zu packen und die fixen Werte durch Verweise auf Zellen eines Excel-Arbeitsblatts wie in Abb. 11.7 zu ersetzen. Das Ergebnis kann etwa so aussehen:

```
Sub Bearbeiten(ss As SAPFEWSELlib.GuiSession)
Dim zeile As Long
zeile = 2
Do While Cells(zeile, 1).Value <> ""
    MaterialBearbeiten ss, zeile
    zeile = zeile + 1
Loop
End Sub

Sub MaterialBearbeiten (ss As SAPFEWSELlib.GuiSession, zeile As Long)
Dim mainw As SAPFEWSELlib.GuiMainWindow
Dim statuszeile As SAPFEWSELlib.GuiStatusbar

Set mainw = ss.FindById("wnd[0]")
mainw.FindById("tbar[0]/okcd").Text = "/nMM02"
mainw.sendVKey 0
mainw.FindById("usr ctxtRMMG1-MATNR").Text = Cells(zeile, 1).Value
mainw.sendVKey 0

If IstMaterialFalsch(ss, zeile) Then
    Exit Sub
End If

ss.FindById("wnd[1]/usr/tblSAPLMGMMTC_VIEW").GetAbsoluteRow(0) _
    .Selected = True
ss.FindById("wnd[1]/usr/tblSAPLMGMMTC_VIEW").GetAbsoluteRow(3) _
    .Selected = True
ss.FindById("wnd[1]/tbar[0]/btn[0]").press

ss.FindById("wnd[1]/usr ctxtRMMG1-WERKS").Text _
    = Cells(zeile, 2).Value
ss.FindById("wnd[1]/usr ctxtRMMG1-VKORG").Text _
    = Cells(zeile, 3).Value
ss.FindById("wnd/usr ctxtRMMG1-VTWEG").Text = Cells(zeile, 4).Value
```

```
ss.FindById("wnd[1]/tbar[0]/btn[0]").press

If IstOrgFalsch(ss, zeile) Then
    Exit Sub
End If

mainw.FindByName("MARA-VOLUM", "GuiTextField").Text _
= Cells(zeile, 5).Value
mainw.FindByName("MARA-VOLEH", "GuiCTextField").Text _
= Cells(zeile, 6).Value
mainw.FindByName("MARA-GROES", "GuiTextField").Text _
= Cells(zeile, 7).Value

mainw.FindById("tbar[0]/btn[11]").press

Abschliessen ss, zeile
End Sub
```

Die Sub Bearbeiten implementiert eine Schleife über die Zeilen des Excel-Arbeitsblatts und damit durch die Materialien. Sie wird aus der Sub Starten aus Abschn. 11.4.1 aufgerufen. Die Sub MaterialBearbeiten besteht aus dem aufgezeichneten und überarbeiteten Code aus dem vorhergehenden Abschnitt und aktualisiert die Daten eines einzelnen Materials. Außerdem protokolliert sie, ob das Material erfolgreich geändert wurde. Wenn Fehler auftreten, protokolliert sie diese ebenfalls und behandelt die Fehlermeldungen, die SAP erzeugt. Die Funktionen und die Sub, die dies umsetzen, beschreibt der folgende Abschnitt.

11.4.5 Fehlerbehandlung

Die Demo-Anwendung „Datenpflege“ soll die gesamte Materialliste bearbeiten, auch wenn bei einzelnen Materialien Fehler auftreten. Damit sie im Fall eines Fehlers weiterlaufen kann, muss sie die Fehlermeldungen quittieren, die SAP erzeugt. Drei Arten von Fehlern können vorkommen. Die unterschiedlichen Behandlungen, die sie erfordern, werden in zwei Funktionen und einer Sub implementiert.

11.4.5.1 Fehler „Falsches Material“

Der erste Fehlerfall besteht darin, dass das zu ändernde Material nicht existiert, siehe Abb. 11.8. Die Statuszeile hat dann den Messagetype „E“. Die Fehlerbehandlungsfunktion bricht die Bearbeitung durch einen Klick auf „Beenden“ ab und verlässt damit die SAP-Transaktion. Zuvor schreibt sie Messagetype und Text der Statuszeile in die Protokollspalten des Excel-Arbeitsblatts. Wenn kein Fehler vorlag, läuft das Skript weiter und trägt dort später andere Werte ein.

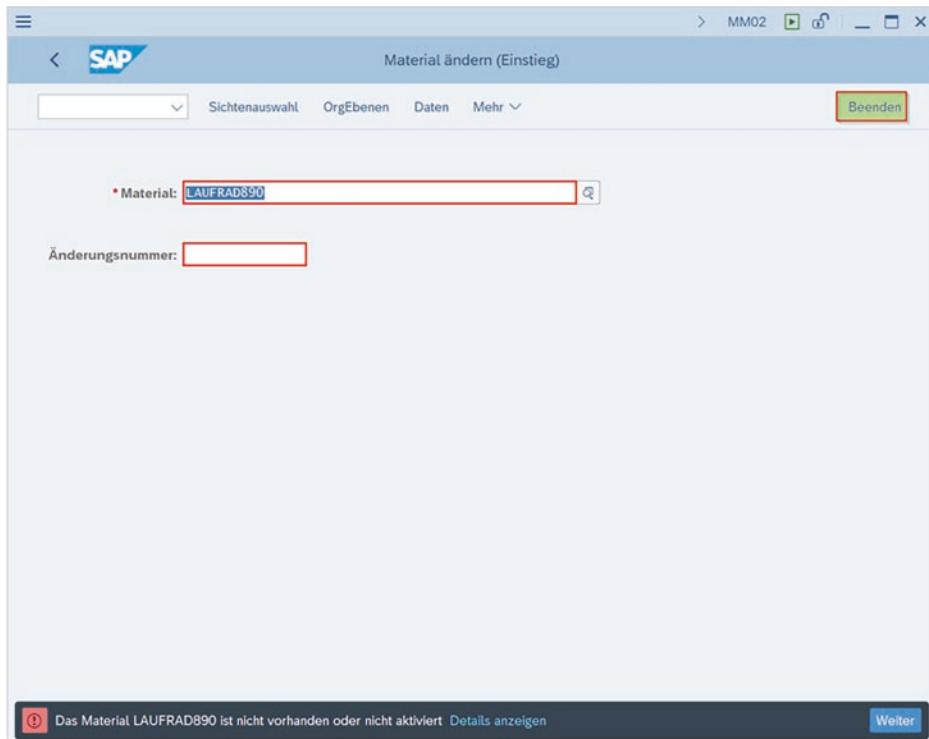


Abb. 11.8 Fehler bei der SAP-Datenpflege: falsches Material

```

Function IstMaterialFalsch(ss As GuiSession, zeile As Long) As Boolean
Dim statuszeile As GuiStatusbar
Set statuszeile = ss.FindById("wnd[0]/sbar")
Cells(zeile, 8).Value = statuszeile.MessageType
Cells(zeile, 9).Value = statuszeile.Text

If statuszeile.MessageType = "E" Then
    ss.FindById("wnd[0]/tbar[0]/btn[15]").press
    IstMaterialFalsch = True
End If
End Function

```

11.4.5.2 Fehler „Falsche Organisationseinheit“

Die zweite Fehlersituation tritt ein, wenn Organisationseinheiten nicht existieren oder nicht zum Material passen. SAP meldet dies in einem Popup-Dialog, siehe Abb. 11.9. Weil auch die Organisationseinheiten in einem Popup-Dialog eingegeben werden, sind damit zwei solche Dialoge offen. Sie erscheinen im Code als wnd[1] und wnd[2]. Die

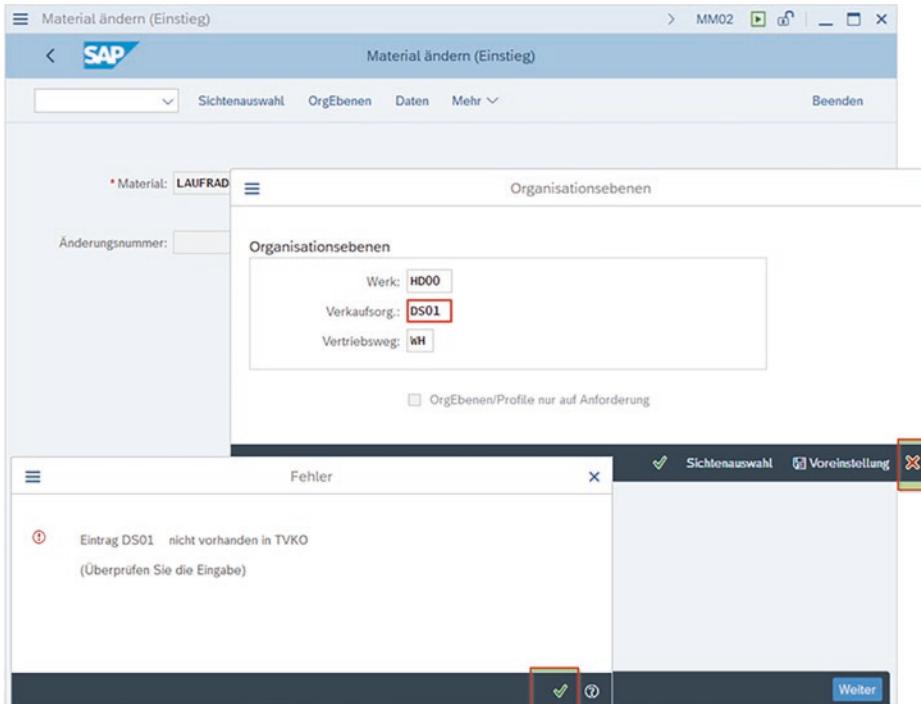


Abb. 11.9 Fehler bei der SAP-Datenpflege: falsche Organisationseinheit

Funktion IstOrgFalsch bestätigt die Fehlermeldung und schließt die Dialogfenster. Zuvor übernimmt sie die Fehlermeldung aus dem Fehler-Popup-Dialog in die Protokollspalte des Excel-Arbeitsblatts und ergänzt dort noch ein „E“ für „Error“.

```
Function IstOrgFalsch(ss As GuiSession, zeile As Long) As Boolean
Dim statuszeile As GuiStatusbar
If ss.ActiveWindow.Text = "Fehler" Then
    Cells(zeile, 8).Value = "E"
    Cells(zeile, 9).Value = ss.FindById("wnd[2]/usr/txtMESSTXT1").Text
    ss.FindById("wnd[2]/tbar[0]/btn[0]").press
    ss.FindById("wnd[1]/tbar[0]/btn[12]").press
    IstOrgFalsch = True
End If
End Function
```

11.4.5.3 Fehler „Falscher Wert“

Die dritte Art von Fehler tritt beim Sichern auf, wenn das Skript einen unzulässigen Wert eingetragen hat wie in Abb. 11.10. Die Statuszeile zeigt dann eine Fehlermeldung

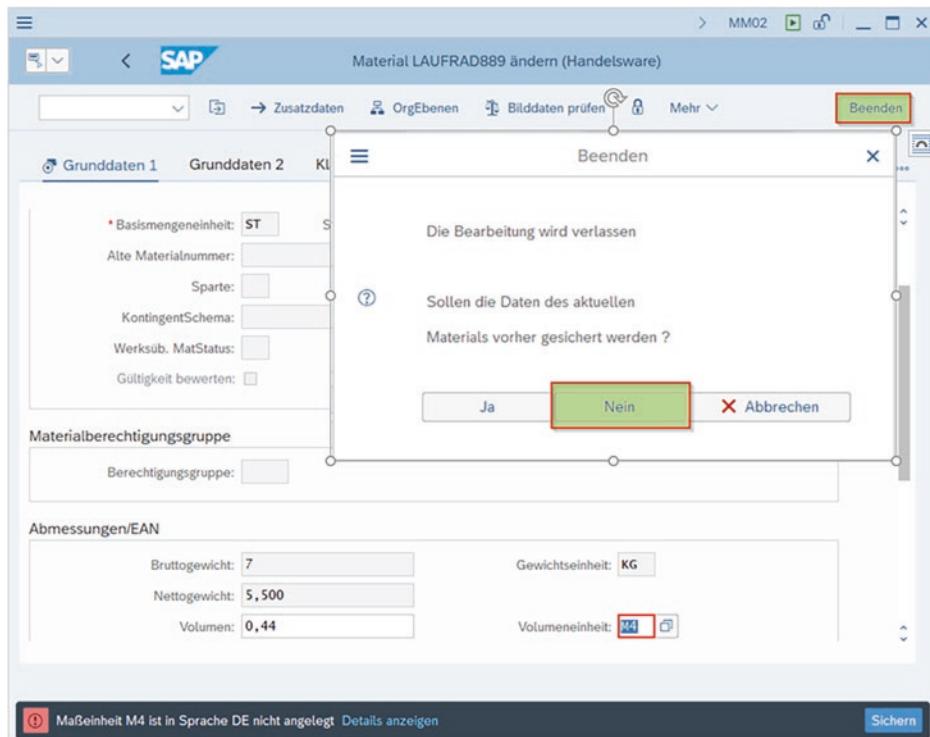


Abb. 11.10 Fehler beim Speichern bei der SAP-Datenpflege

und hat den Messagetype „E“. Sonst zeigt sie eine Erfolgsmeldung und hat den Messagetype „S“. Die Sub Abschliessen übernimmt Messagetype und Meldung in jedem Fall in die Protokollspalten des Excel-Arbeitsblatts. Im Fehler-Fall bricht sie die Materialänderung mit einem Klick auf den Button `btn[15]` „Beenden“ ab und bestätigt anschließend im Popup-Dialog `wnd[1]`, dass die Daten nicht gesichert werden sollen.

```
Sub Abschliessen(ss As GuiSession, zeile As Long)
Dim statuszeile As GuiStatusbar
Set statuszeile = ss.FindById("wnd[0]/sbar")
Cells(zeile, 8).Value = statuszeile.Messagetype
Cells(zeile, 9).Value = statuszeile.Text

If statuszeile.Messagetype = "E" Then
    ss.FindById("wnd[0]/tbar[0]/btn[15]").press
    ss.FindById("wnd[1]/usr/btnSPOP-OPTION2").press
```

```
End If  
End Sub
```

Damit ist die Bearbeitung dieses Materials abgeschlossen und die Demo-Anwendung kann weitere Materialien bearbeiten.

11.4.6 Zusammenfassung

Insgesamt besteht die Demo-Anwendung „Datenpflege“ aus vier Subs und zwei Funktionen: Sub Starten, Sub Bearbeiten, Sub MaterialBearbeiten, Funktion IstMaterialFalsch, Funktion IstOrgFalsch und Sub Abschliessen. Hauptprozedur ist die Sub Starten.

11.5 Tipps für das weitere Vorgehen

11.5.1 Scrollen

Die SAP GUI lädt große Tabellen nach und nach, um lange Antwortzeiten zu vermeiden. Ein SAP GUI-Skript kann nur mit den Tabellenzeilen arbeiten, die im GUI-Fenster sichtbar sind, denn nur diese sind geladen. Um auf weitere Zeilen zuzugreifen, muss es die Scroll-Operationen nachbilden, mit denen ein menschlicher Nutzer die Daten und Einträge sichtbar machen würde. S. Rossi hat ein Skript erarbeitet, das dies bewerkstelltig [9].

11.5.2 Datentabellen herunterladen

Ein Video von J. Ting zeigt einen einfachen Ansatz, um Datentabellen aus der SAP GUI in Excel zu importieren. Dies ist hilfreich, um regelmäßig wiederkehrende Auswertungen und Reports zu erstellen [10].

11.5.3 GUI-Elemente mit dem Scripting Tracker identifizieren

Der Scripting Tracker ist ein sehr hilfreiches Tool, um Buttons und andere GUI-Elemente zu identifizieren, die in Skripte eingebunden werden sollen. Er greift auf ein geöffnetes SAP GUI-Fenster zu und zeigt die Pfade zu dessen GUI-Elementen an. Abb. 11.3 stammt von diesem Tool. Der Entwickler S. Schnell stellt den Scripting Tracker kostenlos zum Download bereit [5].

11.6 Fazit und Ausblick

SAP GUI Scripting ermöglicht es, mit wenig Aufwand und vorhandenen Mitteln kleine Lösungen zu realisieren, die die Arbeit mit SAP effizienter, angenehmer und fehler-sicherer gestalten. Moderne RPA-Tools nutzen SAP GUI-Skripte, um SAP-Automatisierungen in anwendungsübergreifende Workflows einzubinden. Auch außerhalb solcher Tools bieten SAP GUI-Skripte die Chance, interessante und auch ungewöhnliche Automatisierungen zu realisieren. So lässt beispielsweise C. Varga ein SAP GUI-Skript automatisch Screenshots der SAP GUI erstellen und in ein Word-Dokument einfügen [11]. Diese Lösung kann etwa beim Erstellen von Prozessdokumentationen für Schulungen oder Audits helfen.

Literatur

1. Automation Anywhere, „Using GUI Automation“, *Automation Anywhere Docs*, 23. Juni 2022. <https://docs.automationanywhere.com/bundle/enterprise-v11.3/page/enterprise/topics/aae-client/bot-creator/commands/using-gui-automation.html> (zugegriffen 31. Oktober 2022).
2. R. Goyal, „How RPA Developers Can Kick Start SAP Automation | Community Blog“, *Ui-Path*, 10. Februar 2022. <https://www.uipath.com/community/rpa-community-blog/how-rpa-developers-can-kick-start-sap-automation> (zugegriffen 31. Oktober 2022).
3. kathyos und MSFTMan, „Introduction to SAP GUI-based RPA in Power Automate Desktop“, *Microsoft Learn Power Automate*, 16. Februar 2022. <https://learn.microsoft.com/en-us/power-automate/guidance/rpa-sap-playbook/introduction> (zugegriffen 31. Oktober 2022).
4. SAP, „SAP GUI Scripting API | SAP Help Portal“ https://help.sap.com/docs/sap_gui_for_windows (zugegriffen 12. November 2022).
5. S. Schnell, „Scripting Tracker for SAP GUI Scripting“. Stefan Schnell, 22. Juli 2022. <https://tracker.stschnell.de/> (zugegriffen 11. November 2022).
6. SAP, „SAP GUI Scripting Security Guide“. 2019. https://help.sap.com/doc/97d2d0bc2ed248a4a85a0bec608704f8/760.01/en-US/sap_gui_scripting_sec_guide.pdf (zugegriffen 19. Februar 2022).
7. DevGuru, „WSH >> WScript >> ConnectObject“ <http://www.devguru.com/content/technologies/WSH/wscript-connectobject.html> (zugegriffen 19. Februar 2023).
8. S. Azmi, „How to make Excel VBA monitor SAP GUI Events?“, *SAP Community*, 24. September 2013. <https://answers.sap.com/questions/10307820/how-to-make-excel-vba-monitor-sap-gui-events.html> (zugegriffen 21. Februar 2023).
9. S. Rossi, „excel – Reading text in Table Control – Stack Overflow“, *Stackoverflow*, 9. August 2021. <https://stackoverflow.com/questions/68685911/reading-text-in-table-control/68715008#68715008> (zugegriffen 4. November 2022).
10. J. Ting, „Automate SAP Data Extraction with Excel VBA & SAP GUI Scripting – Minimal Coding Required“, Video, 10. September 2021. <https://www.youtube.com/watch?v=ISDX5LwcVPQ> (zugegriffen 30. Oktober 2022).
11. C. Varga, „Taking SAP screenshots with GUI Scripting“, Video, 7. Oktober 2021. https://www.youtube.com/watch?v=W_J-9t_mWDU (zugegriffen 13. November 2022).



Inhaltsverzeichnis

12.1	VBA in SOLIDWORKS	226
12.2	Das Objektmodell von SOLIDWORKS	227
12.2.1	Versionierte Objekttypen und Methoden	227
12.2.2	Nice to know: Interfaces	228
12.3	Der Objekttyp ModelDoc2	229
12.3.1	Code-Beispiel: Umgang mit Modellen	229
12.3.2	Code-Beispiel: Elemente selektieren	230
12.4	Demo-Anwendung „Volumenkörper“	232
12.4.1	Type Koord zur Speicherung eines Punkts	232
12.4.2	Sub Modellieren	233
12.4.3	Sub DimErzeugen zur Bemaßung der Skizze	235
12.4.4	Funktion DimKoordBerechnen zur Positions berechnung mit Static Variablen	236
12.4.5	Die Demo-Anwendung aus Excel starten	238
12.5	Fazit	239
	Literatur	239

SOLIDWORKS ist eine professionelle und leistungsstarke CAD-Software, die vor allem im Maschinenbau viel genutzt wird. Ihre umfangreiche Objektbibliothek eröffnet vielfältige Möglichkeiten für Automatisierungen. Zum Beispiel können Skripte helfen, Dateien konsistent zu benennen und zu verschlagworten, sie in den vorgesehenen Verzeichnissen abzulegen, besondere Darstellungen auszuleiten oder Massenänderungen in komplexen Konstruktionen durchzuführen. Auch kleine Tools für die Konfiguration von Baugruppen, zum automatisierten Zeichnen von Varianten und vieles mehr lassen sich mit VBA in SOLIDWORKS realisieren.

Der große Funktionsumfang und die Leistungsfähigkeit von SOLIDWORKS spiegeln sich in einem komplexen Objektmodell wider. Dieses Kapitel gibt einen Einstieg in das Objektmodell von SOLIDWORKS und erklärt einige seiner Besonderheiten. Den Umgang mit dem Objektmodell zeigt dann ein SOLIDWORKS-Makro, das einen einfachen Volumenkörper konstruiert. Da in der Dokumentation des SOLIDWORKS-Objektmodells häufig der Begriff „Interface“ vorkommt, erläutert das Kapitel auch, was dieser Begriff im Kontext der Objektorientierten Programmierung bedeutet. Weiterhin zeigt dieses Kapitel, wie man `Static`-Variablen einsetzen kann.

12.1 VBA in SOLIDWORKS

VBA ist sehr gut in SOLIDWORKS integriert. Die VBA-Entwicklungsumgebung und den Makrorecorder findet man in der Bedienoberfläche in der Menüleiste **Extras** im Dropdown-Menü **Makro**. Hier gibt es die Optionen, ein vorhandenes Makro zum Bearbeiten zu öffnen oder ein neues Makro zu erzeugen. Auch der Makrorecorder lässt sich hier aktivieren.

SOLIDWORKS speichert Makros nicht in einem SOLIDWORKS-Dokument oder in der Anwendung, es erzeugt separate Dateien im verschlüsselten Format „.swp“. Um ein solches Makro in lesbaren Textformat zu konvertieren, exportiert man es mit der **Export**-Funktion der SOLIDWORKS-Entwicklungsumgebung oder überträgt es einfach mit Copy&Paste in eine Textdatei, etwa mit der Dateiendung „.bas“ für „BASIC“, oder auch in die VBA-Entwicklungsumgebung von, zum Beispiel, Excel. Um umgekehrt Quellcode aus einer „.bas“- oder anderen Textdatei in ein SOLIDWORKS-Makro umzuwandeln, legt man in SOLIDWORKS ein neues Makro an und importiert die „.bas“-Datei oder fügt den Quellcode per Copy&Paste ein.

Die API von SOLIDWORKS ist mit mehreren hundert Funktionen sehr komplex [1]. Entsprechend umfangreich ist auch die Dokumentation. Einführungen in das Objektmodell von SOLIDWORKS, wie zum Beispiel [2], behandeln oft nur Ausschnitte. Bei der Entwicklung von SOLIDWORKS-Makros beginnt man deshalb gern mit aufgezeichneten Makros und überarbeitet und erweitert sie anschließend [3]. Auch die Code-Beispiele, die man an vielen Stellen in der SOLIDWORKS-Dokumentation [4] findet, sind als Ausgangspunkte gut geeignet, siehe zum Beispiel [5, 6].

Mit dem Makrorecoder aufgezeichnete Skripte haben den Nachteil, dass der Makrorecoder Variablen für SOLIDWORKS-Objekte mit dem allgemeinen Typ `Object` dimensioniert. Wer beim Programmieren von der Code-Completion und ähnlichen Hilfsfunktionen der Entwicklungsumgebung profitieren möchte, muss `Object` in den Variablendeclarationen durch die passenden spezifischen SOLIDWORKS-Objekttypen ersetzen.

12.2 Das Objektmodell von SOLIDWORKS

Zwei wichtige Bestandteile des Objektmodells von SOLIDWORKS sind das Objekt `SldWorks` und der Objekttyp `ModelDoc2`. Der Objekttyp `ModelDoc2` repräsentiert die Dokumente oder Modelle, die mit der Anwendung SOLIDWORKS erstellt und bearbeitet werden. Das Objekt `SldWorks` repräsentiert die Anwendung SOLIDWORKS. Es besitzt ein Feld `ActiveDoc`, in dem es das aktuell aktive Modell speichert. Anders als in den Objektmodellen anderer Anwendungen gibt es im SOLIDWORKS-Objektmodell jedoch keine direkt ansprechbare Collection zur Verwaltung offener Dokumente. Methoden zum Erzeugen, Öffnen und Aktivieren von Dokumenten sind direkt im Objekt `SldWorks` angesiedelt.

Auffällig im SOLIDWORKS-Objektmodell sind viele nummerierte Objekttypen und Methoden sowie Interfaces. Diese Begriffe werden in der Dokumentation zum Objektmodell sehr häufig genannt. Sie werden jetzt vorab geklärt. Dann folgen einige Code-Beispiele. Abb. 12.1 zeigt einen dafür relevanten Teil des SOLIDWORKS-Objektmodells.

12.2.1 Versionierte Objekttypen und Methoden

Das SOLIDWORKS-Objektmodell enthält zahlreiche nummerierte Objekttypen und Methoden. Zum Beispiel gibt es die beiden Objekttypen `ModelDoc` und `ModelDoc2`. Der Objekttyp `SketchSegment` besitzt die Methode `Select` und zusätzlich die Methoden `Select2` bis `Select4`. Diese Nummerierungen zeigen meist unterschiedliche Versionsstände der Typen und Methoden an. Sie entstehen, wenn der Softwarehersteller das Softwareprodukt weiterentwickelt und modifiziert. Ältere Versionsstände werden als

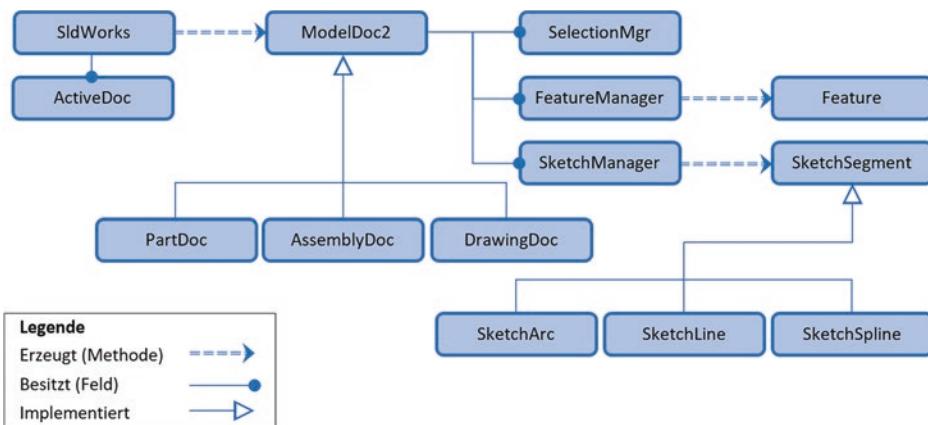


Abb. 12.1 Ausschnitt aus dem SOLIDWORKS Objektmodell, adaptiert und erweitert aus [7, 8]

abgelöst (deprecated) gekennzeichnet, bleiben aber im Objektmodell enthalten, damit bestehende Makros weiterhin funktionieren. Bei der Entwicklung neuer Makros verwendet man dann die aktuellste Version mit der höchsten Nummer.

12.2.2 Nice to know: Interfaces

Die Dokumentation des SOLIDWORKS-Objektmodells nennt häufig den Begriff „Interface“. In der Objektorientierten Programmierung versteht man darunter eine besondere, abstrakte Art Objekttyp, die Eigenschaften und Methoden von anderen, spezielleren (genauer: „implementierenden“) Objekttypen zusammenfassend beschreibt. Abb. 12.1 zeigt zwei Interfaces: `ModelDoc2` und `SketchSegment`. Ihnen sind Objekttypen mit „Implementiert“-Pfeilen zugeordnet. Die Beziehung zwischen einem Interface und seinen implementierenden Objekttypen besteht darin, dass die implementierenden Objekttypen alle Felder und Methoden besitzen, die der Interface-Typ vorgibt, und darüber hinaus noch weitere eigene Felder und Methoden.

Der Zweck von Interfaces ist, Objekte unterschiedlichen Typs im Code einheitlich behandeln zu können, zum Beispiel in einer gemeinsamen Collection oder als Rückgabewert einer Funktion. Eine Collection wird dazu als eine Collection mit Elementen des Interface-Typs definiert. Eine Funktion wird so definiert, dass sie ein Objekt des Interface-Typs als Ergebnis liefert. Aus Sicht der Programmierung wirkt der Interface-Typ quasi als Platzhalter für Objekte der implementierenden Objekttypen. Der Code operiert mit den Eigenschaften und Methoden, die der Interface-Typ definiert. In anderen Code-teilen oder spätestens bei der Ausführung des Codes nimmt dann ein Objekt eines speziellen Objekttyps den Platz des Interface-Typs ein. Dies funktioniert sicher, weil das Objekt des implementierenden Typs alle Eigenschaften und Methoden besitzt, die der Interface-Typ vorgibt und deshalb im Code wie ein Objekt des Interface-Typs behandelt werden kann.

Einige objektorientierte Programmiersprachen bieten Sprachmittel, um Interfaces explizit als solche zu deklarieren. In VBA dagegen erscheinen Interface-Typen wie gewöhnliche Objekttypen. Da die Dokumentation des SOLIDWORKS-Objektmodells neben VBA auch andere Programmiersprachen abdeckt, in denen Interfaces explizit deklariert werden, hebt sie die Interfaces besonders hervor.

Auch im Objektmodell von PowerPoint gibt es einen Objekttyp, der wie ein Interface eingesetzt werden kann, nämlich `Shape`, siehe Kap. 7. Im Unterschied dazu enthält das Objektmodell von Outlook keinen Objekttyp „Item“. Deshalb arbeitet Code, der für verschiedene „Item“-Typen funktionieren soll, mit dem allgemeinen Objekttyp `Object`, wie in Abschn. 8.3.1 erläutert.

12.3 Der Objekttyp ModelDoc2

SOLIDWORKS kennt verschiedene Arten von Modellen und sein Objektmodell stellt dafür passende Objekttypen bereit. Die wichtigsten sind Teil mit dem Objekttyp PartDoc, Baugruppe mit dem Objekttyp AssemblyDoc und Zeichnung mit dem Objekttyp DrawingDoc, wie auch Abb. 12.1 illustriert. Diese Objekttypen implementieren den Interface-Typ ModelDoc2. Die folgenden Code-Beispiele zeigen Möglichkeiten, um mit ihnen umzugehen. Sie laufen in SOLIDWORKS als Wirtsanwendung.

12.3.1 Code-Beispiel: Umgang mit Modellen

Der folgende Code-Schnipsel greift auf ein PartDoc zu, das in SOLIDWORKS geöffnet ist. Die ersten Befehle machen das Objekt SldWorks verfügbar, das die Anwendung SOLIDWORKS repräsentiert. Der Code-Schnipsel weist dann das gerade aktive Dokument aus dem Feld ActiveDoc einer Variablen des Typs PartDoc zu.

Wenn in SOLIDWORKS kein Teil, sondern etwa eine Baugruppe aktiv ist, löst diese Zuweisungsoperation den Laufzeitfehler „Typen inkompatibel“ aus, denn eine Baugruppe lässt sich nicht auf den Objekttyp PartDoc abbilden. Der Code-Schnipsel fängt den Laufzeitfehler ab und gibt eine Meldung aus.

```
Sub DemoPartDoc()
Dim swApp As SldWorks.SldWorks
Set swApp = Application.SldWorks

Dim teil As PartDoc

On Error GoTo NOT_A_PART
Set teil = swApp.ActiveDoc
Debug.Print "Teil"
    ' Tu etwas mit dem Teil ...
Exit Sub

NOT_A_PART:
    MsgBox ("Nur Teil möglich")
End Sub
```

Ein weiteres Programmiermuster für den Umgang mit Interfaces und ihren implementierenden Objekttypen zeigt der folgende Code-Schnipsel. Er verwendet die Methode GetType des Objekttyps ModelDoc2, die den genauen Typ eines ModelDoc2-Objekts in Form einer Nummer angibt, siehe Tab. 12.1. Die Methode GetType ist in allen Arten von Modellen vorhanden. Nachdem der Typ eines Modells bekannt ist, kann es auf seinen speziellen Objekttyp abgebildet werden:

Tab. 12.1 Dokumenttypen im SOLIDWORKS-Objektmodell
[9]

Bedeutung	Nummer
swDOCASSEMBLY	2
swDOCDRAWING	3
swDOCIMPORTED_ASSEMBLY	7; Multi-CAD
swDOCIMPORTED_PART	6; Multi-CAD
swDOCLAYOUT	5
swDOCNONE	0
swDOCPART	1
swDocSDM	4

```

Sub DemoInterfaceType ()
Dim swApp As SldWorks.SldWorks
Set swApp = Application.SldWorks

Dim modell As ModelDoc2
Dim teil As PartDoc
Dim baugruppe As AssemblyDoc

Set modell = swApp.ActiveDoc

Select Case modell.GetType
Case swDocPART:
    Set teil = modell
    Debug.Print "Teil"
    ' Tu etwas mit dem Teil ...
Case swDocASSEMBLY:
    Set baugruppe = modell
    Debug.Print "Baugruppe"
    ' Tu etwas mit der Baugruppe ...
Case Else
    MsgBox ("Nur Teil oder Baugruppe möglich")
End Select
End Sub

```

Eine gut verständliche, etwas ausführlichere Erklärung zu Interfaces und weiterführende Code-Beispiele zum SOLIDWORKS-Objektmodell gibt P. Brinkhuis [10].

12.3.2 Code-Beispiel: Elemente selektieren

Der Objekttyp ModelDoc2 und seine implementierenden Objekttypen sind nicht nur sehr zentral im SOLIDWORKS-Objektmodell, sie besitzen auch sehr viele Eigen-

schaften und Methoden, laut P. Brinkhuis [10] über 700. Weitere Eigenschaften und Methoden sind in ein Extension-Objekt ausgelagert, darunter auch wichtige und häufig benötigte wie zum Beispiel die Methode SelectByID2.

Die Methode SelectByID2 wirkt wie ein Mausklick: ein noch nicht selektiertes Element eines Modells wählt sie an, ein bereits selektiertes Element deselektiert sie. Der Aufruf von Select-Methoden wird in Skripten manchmal nötig, weil einige SolidWorks-Funktionen auf aktuell selektierte Objekte wirken, zum Beispiel die Methode AddDimension2 des Objekttyps ModelDoc2, die in der Demo-Anwendung dieses Kapitels noch zum Einsatz kommt.

Das folgende Demo-Skript verwendet die Methode SelectByID2 der Extension eines PartDoc-Objekts, um zwei Linien „Line1“ und „Line2“ zu selektieren, und greift dann mit dem SelectionManager auf die selektierten Linien zu:

```
Sub DemoSelect()
Dim swApp As Object
Dim SelMgr As SelectionMgr
Dim boolStatus As Boolean
Dim teil As PartDoc
Dim sk As SketchSegment

Set swApp = Application.SldWorks
Set teil = swApp.ActiveDoc

boolStatus = teil.Extension _
    .SelectByID2("Line1", "SKETCHSEGMENT", 0, 0, 0, False, 0, Nothing, 0)
Debug.Print boolStatus

boolStatus = teil.Extension _
    .SelectByID2("Line2", "SKETCHSEGMENT", 0, 0, 0, True, 0, Nothing, 0)
Debug.Print boolStatus

Set SelMgr = teil.SelectionManager
Set sk = SelMgr.GetSelectedObject(1)

Debug.Print sk.GetName
Debug.Print sk.GetLength
sk.Color = vbYellow

Set sk = SelMgr.GetSelectedObject(2)
Debug.Print sk.GetName
Debug.Print sk.GetLength
sk.Color = vbGreen
```

```
sk.DeSelect  
End Sub
```

Beim Selektieren der zweiten Linie ist der sechste Parameter, Append, auf True gesetzt. Dies bewirkt hier, dass die Linie zur selektierten Auswahl hinzugefügt wird. Wäre Append auf False gesetzt, würde die Methode erst die bereits selektierte Auswahl de selektieren und dann die Linie anwählen. Die Parameter und die Verwendungsmöglichkeiten von SelectByID2 sind in der Dokumentation ausführlich beschrieben [11].

Im SOLIDWORKS-Objektmodell gibt es neben SelectByID2 noch diverse weitere Select-Methoden. Sie alle liefern als Rückgabewert nicht etwa ein selektiertes Objekt, sondern einen Booleschen Wert boolStatus, der anzeigt, ob die Select-Methode erfolgreich war. Wenn ja, hat boolStatus den Wert True, sonst den Wert False. Man kann den Rückgabewert boolStatus verwenden, um auf Fehlschläge zu reagieren.

Auch die Demo-Anwendung im folgenden Abschnitt ruft mehrfach Select-Methoden auf. Der Rückgabewert boolStatus wird darin nicht ausgewertet, weil der Code alle selektierten Skizzenelemente selbst erzeugt hat und deshalb sicher sein kann, dass die Select-Aufrufe Erfolg haben.

12.4 Demo-Anwendung „Volumenkörper“

Die Demo-Anwendung modelliert per VBA einen einfachen Volumenkörper. Sie erstellt zuerst eine Kontur aus Linien und rotiert sie dann um die Mittellinie. Die Kontur ist in Abb. 12.2 zu sehen. Den Volumenkörper zeigt Abb. 12.3.

Die Demo-Anwendung läuft in der Wirtsanwendung SOLIDWORKS. Die Liniensegmente der modellierte Kontur werden durch Start- und Endpunkte angegeben, die auch aus einer Excel-Arbeitsmappe gelesen werden könnten. In der Demo-Anwendung sind die Punkte im VBA-Code direkt codiert, damit keine separaten Dateien oder Datenquellen benötigt werden. Insgesamt besteht der Code der Demo-Anwendung aus einer Typ-Definition, zwei Subs und einer Funktion. Der gesamte VBA-Code ist in einem einzigen VBA-Modul enthalten. Die Inhalte des Moduls werden nun in der Reihenfolge beschrieben, in der sie im Modul aufeinanderfolgen.

12.4.1 Type Koord zur Speicherung eines Punkts

Das Modul beginnt mit der Definition des Typs Koord, der die x- und y-Koordinate eines Punkts speichern kann. Dann deklariert es auf Modulebene eine Variable für die Anwendung SOLIDWORKS.

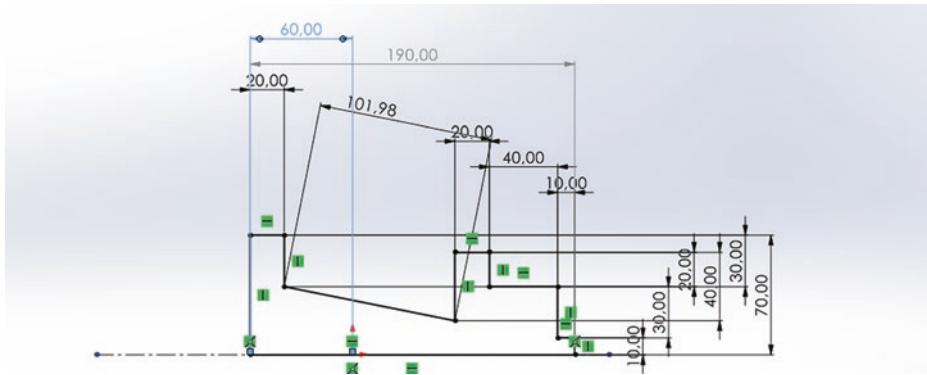


Abb. 12.2 Mit VBA in SOLIDWORKS erzeugte Skizze

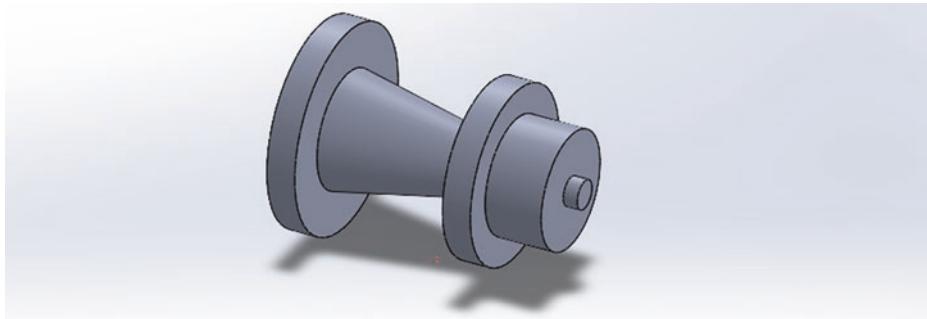


Abb. 12.3 Mit VBA in SOLIDWORKS erzeugter Volumenkörper

```
Type Koord
    x As Double
    y As Double
End Type

Dim swApp As SldWorks.SldWorks
```

12.4.2 Sub Modellieren

Die Sub Modellieren definiert zwölf Punkte, die die Linien der Skizze aufspannen, und weist die Anwendung SOLIDWORKS einer Variablen zu. Dann erzeugt sie mit zwei Befehlen ein neues Teil. Ein Befehl lädt die Standard-Teilvorlage, die für jeden Benutzer angelegt ist. Ein weiterer Befehl erzeugt damit ein Objekt des Typs ModelDoc2:

```

Sub Modellieren()

Dim xy(0 To 12) As Koord
xy(0).x = 0: xy(0).y = 0#
xy(1).x = 0.13: xy(1).y = 0#
xy(2).x = 0.13: xy(2).y = 0.01
xy(3).x = 0.12: xy(3).y = 0.01
xy(4).x = 0.12: xy(4).y = 0.04
xy(5).x = 0.08: xy(5).y = 0.04
xy(6).x = 0.08: xy(6).y = 0.06
xy(7).x = 0.06: xy(7).y = 0.06
xy(8).x = 0.06: xy(8).y = 0.02
xy(9).x = -0.04: xy(9).y = 0.04
xy(10).x = -0.04: xy(10).y = 0.07
xy(11).x = -0.06: xy(11).y = 0.07
xy(12).x = -0.06: xy(12).y = 0#

Dim modell As SldWorks.ModelDoc2
Dim defaultTemplate As String

Set swApp = Application.SldWorks
defaultTemplate = swApp.GetUserPreferenceStringValue(
    swUserPreferenceStringValue_e.swDefaultTemplatePart)
Set modell = swApp.NewDocument(defaultTemplate, 0, 0, 0)

```

Anschließend wird das Teil skizziert. Dies erledigt das SketchManager-Objekt von SOLIDWORKS. Zuerst zeichnet es mit der Methode CreateCenterLine eine Mittellinie und anschließend mit der Methode CreateLine Linien, die die vorab definierten Punkte verbinden. Jede Linie wird sofort bemaßt durch die Sub DimErzeugen, die weiter unten erklärt wird. Eine letzte Linie schließt die Kontur. Diese soll später als Rotationsachse dienen und wird dafür in der Variablen linie gespeichert.

```

Dim sketchMgr As SldWorks.SketchManager
Dim linie As SldWorks.SketchLine
Dim i As Integer
Set sketchMgr = modell.SketchManager
With sketchMgr
    .CreateCenterLine 0.15, 0#, 0#, -0.15, 0#, 0#
    For i = 1 To 11
        .CreateLine xy(i).x, xy(i).y, 0#, xy(i + 1).x, xy(i + 1).y, 0#
        DimErzeugen modell, xy(i), xy(i + 1)
    Next i
    Set linie = .CreateLine(xy(i).x, xy(i).y, 0#, xy(1).x, xy(1).y, 0#)
    DimErzeugen modell, xy(i), xy(1)
End With

```

Um die Skizze vollständig zu definieren, fixiert die Sub Modellieren den Abstand zwischen einem Punkt der Kontur und dem Ursprung. Hierfür bietet sich der Startpunkt der zuletzt gezeichneten Linie an. Im folgenden Codeabschnitt selektiert die Sub diesen Punkt und den Ursprung und erzeugt die Bemaßung.

```
Dim fixpunkt As SldWorks.SketchPoint
Set fixpunkt = linie.GetEndPoint2

fixpunkt.Select4 False, Nothing
modell.Extension.SelectByID2 "Point1@Ursprung", _
    "EXTSKETCHPOINT", 0, 0, 0, True, 6, Nothing, 0
DimErzeugen modell, xy(0), xy(i)
```

Damit ist die Skizze fertig. Der Befehl ViewZoomtofit2 passt die Anzeige der Skizze an. Bevor aus der Kontur ein Volumenkörper entsteht, wartet die Sub auf das Ok einer MessageBox und schafft damit Gelegenheit, die fertige Skizze vorher noch zu betrachten. Diese sollte aussehen wie Abb. 12.2. Danach wird die Kontur durch Rotation zum Volumenkörper.

```
modell.ViewZoomtofit2
If MsgBox("Rotation anwenden?", vbYesNo) = vbYes Then
    linie.Select4 False, Nothing
    Dim rotationFeature As SldWorks.Feature
    Set rotationFeature = _
        modell.FeatureManager.FeatureRevolve2(True, True, False, _
            False, False, False, 0, 0, 6.2831853071796, _
            0, False, False, 0.01, 0.01, 0, 0, 0, True, True, True)
    modell.ShowNamedView2 "*Trimetrisch", 8
    modell.ViewZoomtofit2
End If
DimKoordBerechnen xy(0), xy(0)
End Sub
```

12.4.3 Sub DimErzeugen zur Bemaßung der Skizze

Die Sub DimErzeugen benutzt die Methode AddDimension2 des Objekttyps ModelDoc2, die auf die aktuell selektierte Linie wirkt. Direkt nach ihrer Erzeugung ist eine neu gezeichnete Linie automatisch selektiert. Daher wird die Sub DimErzeugen möglichst unmittelbar nach dem Zeichnen einer Linie ausgeführt. Die Methode AddDimension2 benötigt als Eingabeparameter die Koordinaten der Stelle, an der sie die Bemaßung platzieren soll. Diese Koordinaten berechnet die Funktion DimKoordBerechnen.

```

Sub DimErzeugen (modell As SldWorks.ModelDoc2, p1 As Koord,
                 p2As Koord)
Dim dimKoord As Koord

swApp.SetUserPreferenceToggle _
    swUserPreferenceToggle_e.swInputDimValOnCreate, False
dimKoord = DimKoordBerechnen(p1, p2)
modell.AddDimension2
dimKoord.x, dimKoord.y, 0
swApp.SetUserPreferenceToggle _
    swUserPreferenceToggle_e.swInputDimValOnCreate, True
End Sub

```

Vor dem Aufruf der Methode AddDimension2 wird eine Benutzereinstellung auf False gestellt, um den Popup-Dialog zu unterdrücken, in dem ein Benutzer den Dimensionswert manuell eingeben könnte. Danach wird die Einstellung wieder zurückgesetzt.

Ein Hinweis: Laut SOLIDWORKS API Help ist die Methode AddDimension2 nur zu verwenden, wenn das Modell sichtbar ist [12]. Die Demo-Anwendung und ähnlicher Code können also nicht unsichtbar im Hintergrund ausgeführt werden.

12.4.4 Funktion DimKoordBerechnen zur Positions berechnung mit Static Variablen

Die Funktion DimKoordBerechnen berechnet die Position, an der eine Bemaßung in der Skizze erscheinen soll. Als Ergebnis liefert die Funktion die Koordinaten eines Punkts. Eingabe für die Funktion sind die beiden Punkte, deren Abstand bemaßt werden soll. Zum Beispiel ist in Abb. 12.2 der Abstand zwischen den beiden Punkten, die auf der Mittellinie am weitesten links liegen, mit „60,00“ bemaßt. Der Text „60,00“ dieser Maßangabe wird oben links angezeigt. Die Position dieses Texts und auch der anderen Maße berechnet die Funktion DimKoordBerechnen mit einer einfachen Logik: Die Bemaßung einer eher waagrechten Linie positioniert sie oberhalb der Skizze, die Bemaßung einer eher senkrechten Linie positioniert sie rechts. Ob eine Linie eher senkrecht oder eher waagrecht verläuft, entscheidet die Logik, indem sie den Abstand zwischen den x-Koordinaten mit dem Abstand zwischen den y-Koordinaten der beiden Endpunkte vergleicht. Damit sich die Beschriftungen möglichst wenig überlappen und lesbar bleiben, rückt die Logik jede dazukommende waagrechte Beschriftung etwas weiter nach oben und jede dazukommende senkrechte Beschriftung etwas weiter nach rechts. Ein Offset von 0.015 funktioniert hier gut.

Bei jedem Aufruf muss die Funktion den Offset auf die Position der letzten Bemaßung aufaddieren, damit die Beschriftungen gestaffelt erscheinen. Die jeweils letzten maximalen Positionen speichert die Funktion in der als `Static` deklarierten Variablen `maxkoord`. Dies erspart eine Variable auf Modulebene und ergibt eine in sich

abgeschlossene Funktion und damit gutes Code-Design. Eine Static-Variable behält ihren Wert über die Ausführung der Funktion hinaus, nämlich bis das Modul verlassen oder neu gestartet wird. Deshalb benötigt die Funktion eine Möglichkeit, die Static-Variable zurückzusetzen, wenn eine Skizze fertig gezeichnet ist. In der Sub Modellieren löst ein Funktionsaufruf mit zwei gleichen Eingabewerten den Reset aus. Dieser setzt die gespeicherten Maximalpositionen auf 0. Der letzte Befehl der Sub Modellieren in Abschn. 12.4.2 triggert diesen Reset.

```
Function DimKoordBerechnen(p1 As Koord, p2 As Koord) As Koord
Static maxkoord As Koord
Dim abstandx As Double
Dim abstandy As Double
Dim erg As Koord
Dim OFFSET as Double
OFFSET = 0.015

' reset
If p1.x = p2.x And p1.y = p2.y Then
    maxkoord.x = 0
    maxkoord.y = 0
End If

' init
If maxkoord.x = 0 Then
    maxkoord.x = 0.14
End If
If maxkoord.y = 0 Then
    maxkoord.y = 0.08
End If

abstandx = p1.x - p2.x
abstandy = p1.y - p2.y

If (Abs(abstandx) > Abs(abstandy)) Then
    maxkoord.y = maxkoord.y + OFFSET
    erg.x = p1.x - abstandx/2
    erg.y = maxkoord.y
Else
    maxkoord.x = maxkoord.x + OFFSET
    erg.x = maxkoord.x
    erg.y = p1.y - abstandy/2
End If
DimKoordBerechnen = erg
End Function
```

Bei ihrem ersten Aufruf initialisiert die Funktion die maximalen Positionen mit Startwerten, die außerhalb der entstehenden Skizze liegen. Der erste Aufruf lässt sich daran erkennen, dass die beiden maximalen Positionen noch auf 0 stehen. Um beliebige Skizzen zu bemassen, müsste man die Startwerte geeignet berechnen. Damit der Code der Demo-Anwendung nicht zu lang wird, sind sie hier fix vorgegeben.

12.4.5 Die Demo-Anwendung aus Excel starten

Der gezeigte Code der Demo-Anwendung ist für die Wirtsanwendung SOLIDWORKS vorgesehen. Um ihn aus Excel-VBA zu starten, sind die Verweise in Abb. 12.4 zu setzen. Im Code der Sub Modellieren genügt eine kleine Änderung am Zugriff auf die Anwendung SOLIDWORKS:

```
' Set swApp = Application.SldWorks    ' wird ersetzt durch

Set swApp = CreateObject("SldWorks.Application")
swApp.Visible = True
```

Diese beiden Befehle starten SOLIDWORKS neu und bringen es in den Vordergrund.

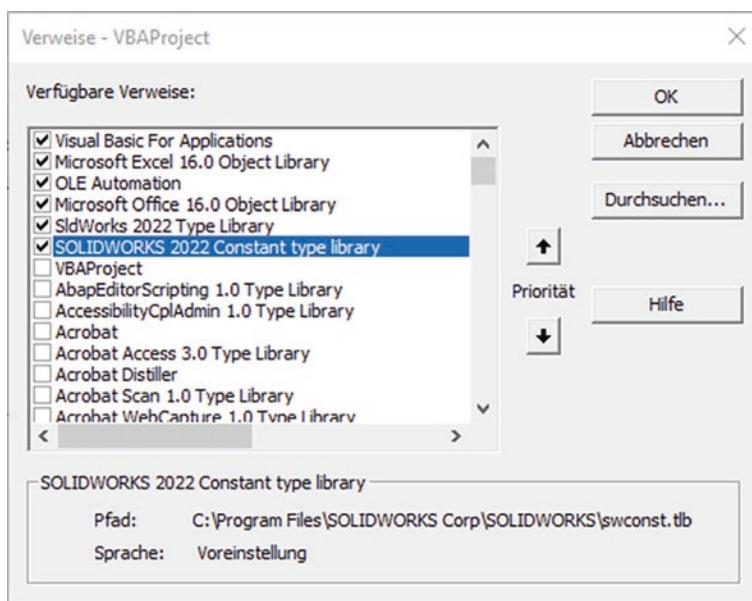


Abb. 12.4 Verweise auf die SOLIDWORKS Bibliotheken in Excel-VBA

12.5 Fazit

Die Demo-Anwendung „Volumenkörper“ automatisiert eine einfache Aufgabe, die mit SOLIDWORKS zu erledigen ist, und verwendet dabei einen kleinen Teil des SOLIDWORKS-Objektmodells. Code-Beispiele auf den SOLIDWORKS-Hilfeseiten und in diversen Online-Makrosammlungen geben weitere Hinweise und Anregungen, zum Beispiel [13]–[15].

Literatur

1. Dassault Systèmes, „SOLIDWORKS API – 2023 – SOLIDWORKS Help“, *SOLIDWORKS Help*, 2022. https://help.solidworks.com/2023/english/SolidWorks/sldworks_c_solidworks_api.htm (zugegriffen 23. Oktober 2022).
2. P. Brinkhuis, „The SOLIDWORKS Object Model + API explained (part 1)“, *CAD Booster*, 1. Dezember 2021. <https://cadbooster.com/the-solidworks-object-model-api-explained-part-1/> (zugegriffen 23. Oktober 2022).
3. S. Petrock, „It's Not Magic! It's a Macro: Huge Time Savings in SOLIDWORKS“, *Engineers Rule*, 31. August 2021. <https://www.engineersrule.com/its-not-magic-its-a-macro-huge-time-savings-in-solidworks/> (zugegriffen 23. Oktober 2022).
4. Dassault Systèmes, „Getting Started – 2023 – SOLIDWORKS API Help“. https://help.solidworks.com/2023/english/api/SWHelp_List.html?id=ee284c0dfd724e7e9fa9f833ef8ea421#Pg0 (zugegriffen 7. Juni 2023).
5. Dassault Systèmes, „Get Sketch Segment Length Example (VBA) – 2023 – SOLIDWORKS API Help“. https://help.solidworks.com/2023/english/api/sldworksapi/Get_Sketch_Segment_Length_Example_VB.htm (zugegriffen 7. Juni 2023).
6. Dassault Systèmes, „Change Color of Face Example (VBA) – 2023 – SOLIDWORKS API Help“. https://help.solidworks.com/2023/english/api/sldworksapi/change_color_of_face_example_vb.htm (zugegriffen 7. Juni 2023).
7. B. A. Hasan, J. Wikander, und M. Onori, „Assembly Design Semantic Recognition Using SolidWorks-API“, *IJMERR*, Bd. 5, 2016, doi: <https://doi.org/10.18178/ijmerr.5.4.280-287>.
8. kithong.lee, „SOLIDWORKS API FUNDAMENTALS #4 | Understanding the SolidWorks Object Model Part 2“, *IME Wiki*, 29. Dezember 2020. <https://wiki.cadcam.com.my/knowledgebase/solidworks-api-fundamentals-4-understanding-the-solidworks-object-model-part-2/> (zugegriffen 24. Oktober 2022).
9. Dassault Systèmes, „swDocumentTypes_e Enumeration – 2023 – SOLIDWORKS API Help“. https://help.solidworks.com/2023/english/api/swconst/SOLIDWORKS.Interop.swconst~SOLIDWORKS.Interop.swconst.swDocumentTypes_e.html (zugegriffen 26. Februar 2023).
10. P. Brinkhuis, „SOLIDWORKS API: the basics – SldWorks, ModelDoc2 (part 2)“, *CAD Booster*, 1. Dezember 2021. <https://cadbooster.com/solidworks-api-basics-sldworks-modeldoc2/> (zugegriffen 24. Oktober 2022).
11. Dassault Systèmes, „SelectByID2 Method (IModelDocExtension) – 2023 – SOLIDWORKS API Help“. <https://help.solidworks.com/2023/english/api/sldworksapi/SolidWorks.Interop.sldworks~SolidWorks.Interop.sldworks.IModelDocExtension~SelectByID2.html> (zugegriffen 8. Juni 2023).

12. Dassault Systèmes, „AddDimension2 Method (IModelDoc2)“, *SOLIDWORKS API Help*, 2022. <https://help.solidworks.com/2022/english/api/sldworksapi/solidworks.interop.sldworks~solidworks.interop.sldworks.imodeldoc2~adddimension2.html?verRedirect=1> (zugegriffen 29. Oktober 2022).
13. A. Taturevych, „SOLIDWORKS API Code Examples for Macros and Add-ins“, *CodeStack*. <https://www.codestack.net/solidworks-api/> (zugegriffen 26. Februar 2023).
14. A. Taturevych, „Library of macros and scripts to automate SOLIDWORKS“, *CodeStack*. <https://www.codestack.net/solidworks-tools/> (zugegriffen 23. Oktober 2022).
15. The CAD Coder, „VBA In Solidworks“, *The CAD Coder*, 23. Oktober 2022. <https://thecadcoder.com//Solidworks-macro-guide/> (zugegriffen 23. Oktober 2022).



VBA-Tools unternehmenstauglich gestalten

13

Inhaltsverzeichnis

13.1	Nice to Know: End User Computing in Forschung und Bildung	242
13.2	Nice to Know: End User Computing im Unternehmen – Chancen und Risiken.....	243
13.3	Gestaltungstipps für VBA-Tools	245
13.3.1	Startbildschirm für Excel-basierte Tools	245
13.3.2	Administrative Informationen bereitstellen.....	245
13.3.3	Konfiguration ermöglichen	246
13.3.4	Umgebungsvariablen nutzen	246
13.4	Wartungsfreundliche Programmierung	247
13.4.1	Konstanten verwenden.....	248
13.4.2	VBA-Module benennen	248
13.5	Makros automatisch starten	249
13.5.1	Nice to know: Ereignisgesteuerte Programmierung	249
13.5.2	Ereignisbehandlungsroutinen entwickeln	250
13.5.3	Risiken durch VBA-Makros.....	251
13.6	Ein Makro aus der Symbolleiste für den Schnellzugriff starten	251
13.7	Fazit und Ausblick	253
	Literatur.....	254

Mit der zunehmenden Digitalisierung im privaten Bereich steigen nicht nur die Ansprüche der Mitarbeiter an digitale Tools im beruflichen Umfeld, sondern auch ihre digitale Kompetenz. Die Unternehmen treiben die Digitalisierung voran, doch oft bremsen Kapazitätsengpässe in den IT-Abteilungen. Als Lösung rückt hier das End User Computing in den Fokus: End User entwickeln digitale Tools, die sie später nutzen, selbst. Auch in anderen Bereichen entwickeln Endnutzer mit VBA hilfreiche Anwendungen, wie Beispiele aus renommierten Forschungs- und Bildungseinrichtungen auf der ganzen Welt zeigen.

Doch bei allen Vorteilen und Potenzialen kann End User Computing auch Risiken und Nachteile mit sich bringen. Diese werden vor allem im Kontext der Unternehmen diskutiert. Dieses Kapitel führt in die Diskussion über End User Computing ein. Es nennt Möglichkeiten, um mit einfachen Mitteln seine Risiken und Nachteile zu minimieren, und gibt Tipps und Vorschläge zur Gestaltung von VBA-Tools, die auch für Nicht-ITer leicht umsetzbar sind. Dazu gehört auch eine Einführung in das ereignisbasierte Programmieren mit VBA.

13.1 Nice to Know: End User Computing in Forschung und Bildung

Nicht nur in Unternehmen, auch in der Forschung und im Bildungsbereich entwickeln End User Tools mit VBA. Sie machen damit die praktischen Ergebnisse ihrer Forschung nutzbar und verbreiten sie, unterstützen ihre eigene Arbeit oder erschließen die Funktionalität anderer Software-Tools:

- Am CERN, dem Conseil Européen pour la Recherche Nucléaire nahe Genf, das zu den weltgrößten und renommieritesten Zentren für physikalische Grundlagenforschung gehört, wurde eine Spezial-Software zu Erstellung von Schaltplänen angeschafft. Sie sollte Schaltpläne für die Kryotechnik schneller und einheitlicher als bisher erstellen und den manuellen Arbeitsaufwand der Forscher reduzieren. Der angestrebte Effekt stellte sich jedoch erst ein, nachdem das Forscherteam ein Tool entwickelt hatte, das die Eingabedateien für diese Software automatisch generiert. Als Plattform wählten sie dafür Excel mit VBA, da damit Entwicklung und Wartung besonders einfach waren. [1]
- An der Universität Quebec haben Mitarbeiter der Abteilungen für Systems Engineering und Mechanical Engineering ein Framework für die Layoutplanung von Baustellen geschaffen, das neben qualitativem Regelwissen auch quantitative Daten wie Kosten und räumliche Distanzen auswertet. Datenextraktion und -verarbeitung realisiert dieses Framework mit VBA-Makros. Der in Excel enthaltene Solver stellt den Algorithmus für die Optimierung der Layouts. [2]
- Verfahrenstechniker der renommierten Universität KU Leuven in Belgien haben verschiedene Ansätze evaluiert, um die Prozesssimulationsssoftware Aspen Hysys für den Datenaustausch mit anderen Software-Tools zu verknüpfen. Sie empfehlen dafür Excel-VBA gegenüber Matlab, Python und Unity. [3]
- In Japan haben Forscher einen Simulator für nukleare Brennstoffkreisläufe auf Basis von Excel realisiert (am Laboratory for Zero-Carbon Energy des Tokyo Institute of Technology und am Nuclear Science Research Institute der Japan Atomic Energy Agency). Er ist größtenteils in VBA programmiert, mit Ausnahme einiger Berechnungen, die aus Effizienzgründen in C++ umgesetzt wurden. Seine Entwickler

möchten den Simulator an viele Interessengruppen in Politik, Industrie und Forschung verteilen und haben Excel-VBA als Entwicklungsplattform gewählt, weil es sehr verbreitet und vielen Personen vertraut ist. [4]

- Auch in Bildung und Lehre kommt VBA zum Einsatz. Beispielsweise wird an einer indischen Universität ein damit erstelltes Berechnungstool in der Lehre zur Konstruktion von Maschinenelementen erfolgreich eingesetzt [5].
- An der FH Aachen helfen mit Excel und VBA umgesetzte Berechnungsschemata, Prinzipien der Wärmetechnik und der Verfahrenstechnik nachzuvollziehen und damit verbundene, reale Aufgabenstellungen effektiv zu lösen [6, 7].
- An einer polnischen Universität ermöglicht eine in Excel erstellte Bedienoberfläche Nutzern ohne Programmiererfahrung, mit Python Datenanalysen durchzuführen [8].

Diese Beispiele zeigen viele Vorteile und Möglichkeiten von VBA auf.

13.2 Nice to Know: End User Computing im Unternehmen – Chancen und Risiken

Auch in Unternehmen initiieren, entwickeln und betreiben End User und Fachabteilungen IT-Lösungen. Dies ist in der Praxis schon lange weit verbreitet und kommt in unterschiedlichen Formen vor: Als Schatten-IT bezeichnet man sämtliche geschäftsprozessunterstützenden IT-Systeme und IT-Service-Prozesse, die einzelne User, Abteilungen oder sonstige Organisationseinheiten in Unternehmen ohne Beteiligung und ohne Kenntnis der zentralen Unternehmens-IT entwickeln oder nutzen [9]. Wenn Fachbereiche IT-Lösungen mit Kenntnis und Zustimmung der zentralen Unternehmens-IT initiieren und betreiben, spricht man von Business-managed IT oder fachbereichsgetriebener IT [10]–[12]. Werden solche Lösungen von Endnutzern in den Abteilungen selbst entwickelt, spricht man von End User Computing oder End User Development. Noch weiter geht das Citizen Development. Hier ist das Ziel, dass ganze Gruppen von Nicht-ITlern zur Erstellung von IT-Lösungen beitragen, wofür das Unternehmen geeignete Tools und Plattformen bereitstellt [13].

Eine Studie untersuchte 2015 den Einsatz von fachbereichsgetriebener IT (FAB-IT) bei vier Unternehmen in Deutschland, davon zwei Industrieunternehmen [14]. Das Ergebnis:

- In allen betrachteten Abteilungen war FAB-IT im Einsatz.
- Insgesamt entdeckte die Studie 386 FAB-IT-Lösungen.
- Ungefähr 34 % der FAB-IT war geschäftskritisch, das heißt, notwendig für die Durchführung von wichtigen oder hochwichtigen Geschäftsprozessen.
- In den beiden Industrieunternehmen basierten 20 % der FAB-IT auf Office-Programmen wie Excel, Access oder Outlook.

Früher wurde in der Literatur vor allem die Schatten-IT thematisiert. Dieser Begriff hat einen negativen Beiklang, oft werden im Zusammenhang damit Risiken und Nachteile genannt [9]: Die Entwicklung und der Betrieb binden in den Fachbereichen Ressourcen, die für andere Aufgaben vorgesehen sind. Weil verschiedene Fachbereiche ähnliche Systeme parallel betreiben, gehen Synergien und Skaleneffekte verloren. Systeme der Schatten-IT halten sich nicht an bestehende Compliance- und Sicherheitsanforderungen, verletzen den Datenschutz oder können die Datensicherheit nicht gewährleisten, wobei dies den Betreibern der Schatten-IT nicht ausreichend bewusst ist. Auch kann Schatten-IT gegen die Ziele des Managements und gegen die Unternehmensstrategie arbeiten, Kontrollmechanismen der Unternehmens-IT unterlaufen und deren Position im Unternehmen schwächen.

Neben organisatorischen Nachteilen werden auch technische Risiken befürchtet [9]: Die Lösungen können eine geringe Qualität haben und unzureichend dokumentiert, getestet und gewartet sein. Die Anwender werden abhängig von Einzelpersonen, die als Einzige die Systeme kennen und warten können. Mangelnde Integration mit anderen Softwaresystemen des Unternehmens verhindert automatischen Datenaustausch, Daten liegen redundant in verstreuten Datensilos und es kommt zu Dateninkonsistenzen und Datenfehlern.

In neuerer Zeit treten die Chancen und Vorteile der von Fachanwendern angestoßenen IT-Lösungen stärker in den Vordergrund: Die so erstellten Lösungen sind typischerweise optimal auf die Bedürfnisse und Anforderungen der Anwender abgestimmt. Oft liefern sie Lösungen für kleine Probleme, für die die zentrale IT keine Zeit hat. Sie tragen zur Innovationskraft und Anpassungsfähigkeit des Unternehmens bei, fördern Eigeninitiative und Kreativität der Mitarbeiter und wirken sich positiv auf die Zufriedenheit von Mitarbeitern aus, da sich diese oft stark mit diesen Lösungen identifizieren und durch sie ein Gefühl der Wirksamkeit und Autonomie gewinnen [9, 10]. Nicht zuletzt steigern sie den Unternehmenserfolg, indem sie die Arbeit der Mitarbeiter produktiver machen.

Auch die Fachbereiche erkennen Vorteile in der FAB-IT. In ihrer Wahrnehmung erfüllen die offiziellen IT-Lösungen die Anforderungen oder Wünsche der Anwender oft nicht ausreichend gut, werden gar nicht angeboten oder es dauert zu lange, bis die offizielle Unternehmens-IT Lösungen bereitstellt, etwa wegen Überlastung oder weil sie besonders hohe Ansprüche an die Lösungen richtet im Hinblick auf Sicherheit, Fehlerfreiheit oder Breite der Einsatzmöglichkeiten. [9]

Man kann davon ausgehen, dass an End User Computing heute kein Weg vorbeiführt. Nicht nur wegen seiner Vorteile, sondern auch, weil Fachbereiche ihre IT-Lösungen als Schatten-IT entwickeln, wenn Unternehmen und zentrale IT das End User Development nicht unterstützen [13]. Bei der Schatten-IT befürchtete Risiken bestehen gleichermaßen auch beim offiziell autorisierten End User Computing und Citizen Development. Wenn ein Unternehmen Citizen Development erfolgreich etabliert, muss es auch entsprechende professionelle IT-Ressourcen für die Wartung der Software und zur Unterstützung der entwickelnden End User bereitstellen [13]. Insgesamt wird erwartet, dass die Fachabteilungen zukünftig noch mehr Verantwortung für IT-Projekte und IT-Lösungen an sich ziehen werden [12, 15].

Vom Unternehmen autorisiertes End User Development und auch nicht-autorisierte Schatten-IT haben die gute Absicht, die Arbeit der Mitarbeiter für das Unternehmen produktiver und effizienter zu machen. Mit einfachen Maßnahmen können Entwickler in den Fachbereichen Risiken und Nachteile von VBA-basierter FAB-IT reduzieren und dafür sorgen, dass die entwickelten Lösungen leichter zu warten und verwalten sind.

13.3 Gestaltungstipps für VBA-Tools

Bei der Digitalisierung mit VBA geht es darum, kleine Anwendungen und Tools auf Basis von Office- und anderer Software zu entwickeln. Diese Anwendungen sind Desktop-Anwendungen (im Unterschied zu Cloud-basierten Lösungen). Sie werden als Dateien verbreitet und kommen mit der Zeit möglicherweise in verschiedenen Versionen im Umlauf. Die folgenden Vorschläge und Tipps sorgen mit wenig Aufwand für eine professionelle Wirkung eines Tools und reduzieren die Risiken von FAB-IT.

13.3.1 Startbildschirm für Excel-basierte Tools

Ein Startbildschirm verleiht einem Excel-basierten Tool ein professionelles Aussehen und lässt sich auf dem ersten Arbeitsblatt schnell einrichten. Er kann zum Beispiel Schaltflächen für die Hauptfunktionen der Anwendung anbieten, Infos über das Tool anzeigen oder auf weitere Arbeitsblätter verlinken. Große Schaltflächen ergeben ein modernes Kachel-Design, vor allem wenn man die Gitternetzlinien des Arbeitsblatts ausschaltet. Ordnet man die Schaltflächen zu sinnvollen Gruppen oder zum Beispiel in der Reihenfolge, die dem typischen Arbeitsablauf entspricht, lässt sich das Tool intuitiver bedienen.

13.3.2 Administrative Informationen bereitstellen

Einige der bekannten Risiken von FAB-IT lassen sich schon dadurch verringern, dass man einem VBA-Tool Informationen für die Administration mitgibt. Dies kann zunächst ganz einfach bedeuten, dass das Tool einen aussagekräftigen und leicht erinnerbaren Namen bekommt. Hilfreich sind weiterhin eine kurze Beschreibung, die den Zweck der Anwendung erklärt, Versionsnummern mit Erstellungsdatum und die Daten einer Ansprechperson, die bei Problemen kontaktiert werden kann. Es gibt mehrere Optionen, um solche Administrationsinformationen bereitzustellen.

Bei einem Excel-basierten Tool, das einen Startbildschirm besitzt, sind weniger umfangreiche administrative Informationen dort gut platziert. Auch eine MessageBox, die sich über eine entsprechende Schaltfläche auf dem Startbildschirm öffnen lässt, eignet sich.

Ausführlichere Informationen und eventuell auch eine Bedien-Hilfe finden auf eigenen Arbeitsblättern Platz. Wenn man ihre Arbeitsblattregister einfärbt und geeignet beschriftet („About“, „Über“, „Info“, „Hilfe“), wird ihre besondere Rolle für Benutzer deutlich. Auch die übrigen Arbeitsblätter eines Excel-Tools sollten passende Bezeichner bekommen, die die automatisch vergebenen, wenig aussagekräftigen Bezeichner „Tabelle 1“, „Tabelle 2“ usw. ersetzen.

Bei Tools, die nicht auf Excel basieren, können die Dokumenteigenschaften administrative Informationen aufnehmen. Eine weitere Option ist, bei jedem Start automatisch eine MessageBox mit solchen Informationen zu anzeigen. Wie sich dies mit dem ereignisorientierten Programmieransatz von VBA umsetzen lässt, zeigt ein einfaches Code-Beispiel für Word in Abschn. 13.3.3. In Office-Anwendungen kann man auch eine Schaltfläche in die **Symbolleiste für den Schnellzugriff** integrieren, die eine solche MessageBox öffnet. Diese Option wird in Abschn. 13.6 beschrieben.

13.3.3 Konfiguration ermöglichen

Wenn sich der Anwendungskontext einer Lösung ändert oder neue User oder Anwendungsbereiche dazukommen, sind oft Anpassungen nötig. Vielleicht müssen neue Verzeichnispfade, Benutzeradressen, Maximal- oder Minimalwerte, Faktoren für Berechnungen oder Ähnliches eingepflegt werden. Schon bei der Entwicklung eines Tools kann man dafür sorgen, dass solche Anpassungen einfach und fehlerfrei vonstatten gehen.

Ein Tool wird ohne Eingriff in den VBA-Code konfigurierbar, wenn man Daten, die von einer Konfiguration betroffen sein können, nicht fest im Code hinterlegt, sondern über eine geeignete Maske editierbar macht. In einem Excel-basierten Tool kann ein spezielles Arbeitsblatt mit der Bezeichnung „Konfiguration“ oder „Einstellungen“ solche Daten enthalten. Der VBA-Code liest die Einstellungen dann von dort ein. Die Demo-Anwendung zu PowerPoint im Abschn. 7.2 verwendet diesen Ansatz. Ein Blattschutz mit Passwort kann bei Bedarf solche Infos vor versehentlichen Änderungen schützen. Auch UserForms oder Dokumenteigenschaften können solche Einstellungen sichtbar und editierbar machen. Sie eignen sich auch für Tools, die nicht in Excel, sondern in Word, PowerPoint etc. laufen. Ein weitere Möglichkeit, Benutzereinstellungen zu speichern, eröffnen die Umgebungsvariablen.

13.3.4 Umgebungsvariablen nutzen

Die VBA-Funktion `Environ$` ruft Systeminformationen aus den Umgebungsvariablen ab. Interessant ist die Umgebungsvariable „username“, die den aktuell angemeldeten Windows-Benutzer speichert. Wenn beispielsweise eine von mehreren Personen genutzte Anwendung Benutzeraktivitäten in Protokollen und Log-Dateien aufzeichnet, kann damit auch der Name des jeweiligen Benutzers protokolliert werden.

Andere Umgebungsvariablen speichern Pfade zu Dateiverzeichnissen, die Windows für jeden Benutzer automatisch anlegt, darunter das Heimverzeichnis und die Verzeichnisse für Anwendungsdaten („AppData“). Letztere sind ein Ablageort für benutzerindividuelle Vorlagen und ähnliche übergreifend benötigte Daten der Softwareanwendungen. Word zum Beispiel speichert hier persönliche Wörterbücher, das Master-Literaturverzeichnis und selbst erstellte Tabellen- und Diagrammvorlagen (Templates), die ständig und nicht nur für ein spezifisches Word-Dokument verfügbar sein sollen. Outlook speichert hier auch das Outlook-VBA-Projekt eines Benutzers, wie in Abschn. 9.1 beschrieben.

Der Vorteil dieser automatisch angelegten Dateiverzeichnisse ist, dass sie auf allen User-Rechnern existieren und dass ihre Pfade für alle User in einheitlich bezeichneten Umgebungsvariablen zu finden sind. Code, der mittels der Umgebungsvariablen auf diese Verzeichnisse zugreift, funktioniert also in der Regel für beliebige Benutzer zuverlässig und ohne individuelle Anpassungen.

Für die Verwendung der Anwendungsdaten-Verzeichnisse gibt es eine gute Praxis: Jede Anwendungssoftware benutzt ein eigenes Unterverzeichnis der AppData, das so heißt wie die Anwendungssoftware selbst. Wenn mehrere Anwendungen eines Anbieters oder Entwicklers die AppData benutzen, wird dort ein Unterverzeichnis mit dem Anbieter- oder Entwicklernamen angelegt und darunter für jede dieser Anwendungen ein eigenes Unterverzeichnis.

Der folgende Code-Schnipsel demonstriert zwei Arten, um auf Umgebungsvariablen zuzugreifen, nämlich per Index und per Name.

```
Sub UmgebungsvariablenDemo ()  
Dim i As Integer  
For i = 1 To 55  
    Debug.Print i & ": " & Environ$(i)  
Next i  
Debug.Print Environ$("USERNAME")  
Debug.Print Environ$("HOMEPATH")  
Debug.Print Environ$("APPDATA")  
End Sub
```

Die Schleife listet die Bezeichner und Werte von bis zu 55 Umgebungsvariablen auf. Die darauf folgenden drei Befehle zeigen, wie man mit dem Bezeichner den Wert einer bestimmten Umgebungsvariablen abrufen kann.

13.4 Wartungsfreundliche Programmierung

Auch bei der Entwicklung machen schon einfache Maßnahmen VBA-Tools wartungsfreundlicher.

13.4.1 Konstanten verwenden

In manchen Fällen benötigt ein VBA-Tool Einstellungen, die man nicht in einer Konfigurationsmaske oder Ähnlichem offenlegen möchte, etwa einen API-Key. Möglicherweise sind Änderungen auch nur in seltenen Ausnahmefällen zu erwarten oder betreffen nur einen einzelnen Wert, sodass ein Konfigurationsblatt oder eine UserForm zu aufwendig erscheinen. Dann ist es günstig, solche Werte im Code als Konstanten zu definieren und die Konstanten im Code so zu platzieren, dass sie leicht zu finden sind, etwa im Hauptmodul oder in einem eigenen Modul „Einstellungen“ des VBA-Projekts wie in Abb. 13.1. Zu ändernde Codestellen sind so leicht zu finden und Änderungen bleiben auf eine einzige Codestelle beschränkt, auch wenn der geänderte Wert an mehreren Stellen im Code verwendet wird.

13.4.2 VBA-Module benennen

Wenn man VBA-Module sinnvoll benennt, findet man sich auch nach längerer Zeit wieder leichter im VBA-Projekt zurecht. Dazu editiert man die Modulbezeichnung im **Eigenschaften**-Fenster der VBA-Entwicklungsumgebung, siehe Abb. 13.1. Leider lassen sich die VBA-Module nicht beliebig anordnen, die VBA-Entwicklungsumgebung listet sie immer in alphabetischer Reihenfolge auf. Hier hat sich bewährt, das zentrale Modul mit den wichtigsten Makros einheitlich zu bezeichnen, zum Beispiel als „Main“. So fällt

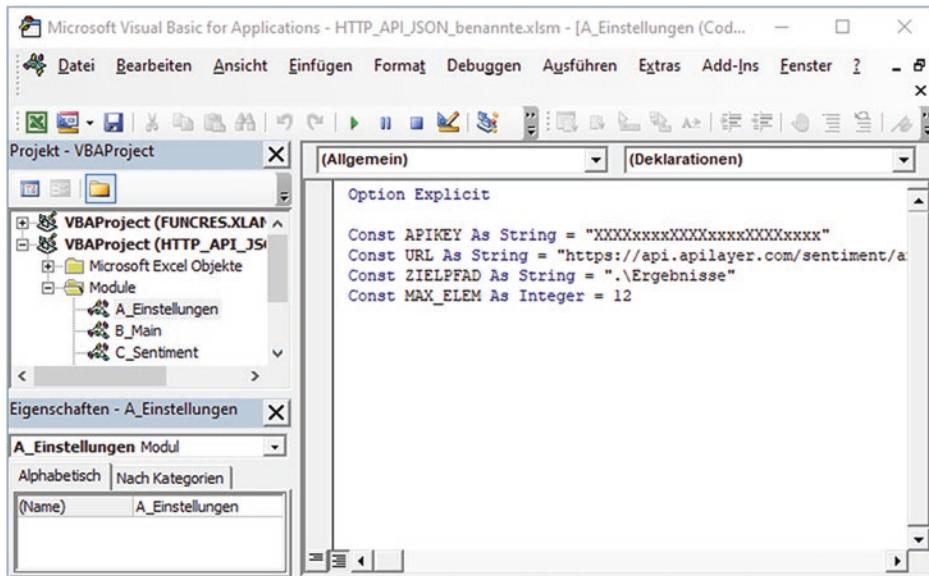


Abb. 13.1 Module im Eigenschaften-Fenster benennen und anordnen

der Einstieg in das VBA-Projekt leichter, wenn die ursprünglichen oder neuen Entwickler es nach einiger Zeit anpassen, erweitern oder Fehler beheben. Bei umfangreicheren VBA-Projekten kann man die Module mit vorangestellten Buchstaben-Indizes in eine bestimmte Reihenfolge bringen wie in Abb. 13.1.

13.5 Makros automatisch starten

Als ereignisgesteuerte Programmiersprache macht VBA es einfach, Makros in bestimmten Situationen automatisch zu starten, die zum Beispiel beim Öffnen eines Dokuments eine MessageBox anzeigen oder beim Speichern Schlagwörter und weitere Metainformationen in die Dokumenteneigenschaften einpflegen.

13.5.1 Nice to know: Ereignisgesteuerte Programmierung

In der ereignisgesteuerten Programmierung bildet ein Ereignis ein Vorkommnis ab, das losgelöst vom Kontrollfluss der laufenden Anwendung stattfindet. Es wird als Ereignis modelliert, damit die Anwendung mit einer ihrer Komponenten darauf reagieren kann. Ein solches Ereignis kann eine Benutzeraktion sein, wie etwa ein Mausklick oder eine Tastatureingabe, oder das Öffnen oder Schließen der Anwendung oder einer von ihr bearbeiteten Datei. Viele derartige Ereignisse sind im Objektmodell einer mit VBA automatisierbaren Anwendung bereits angelegt. Darüber hinaus lassen sich auch weitere, eigene Ereignisse definieren, beispielsweise Ereignisse, die anzeigen, dass eine Frist abgelaufen ist, ein Schwellwert überschritten wurde, ein Währungs- oder Aktienkurs sich geändert hat, eine neue Datei in einem bestimmten Ablageverzeichnis abgelegt wurde, und vieles mehr.

Das Gegenstück zu Ereignissen sind Programmteile, die auf ein Ereignis reagieren, die sogenannten Ereignisbehandlungs Routinen (event handler). In VBA und auch in anderen objektorientierten Programmiersprachen sind Ereignisse und Ereignisbehandlungs Routinen Klassen zugeordnet (siehe zu Klassen auch Abschn. 3.1). Dabei kann eine Klasse die Rolle eines Ereignis-Erzeugers oder eines Ereignis-Empfängers übernehmen.

Klassen, die als Ereignis-Empfänger wirken, besitzen Ereignisbehandlungs Routinen, um Ereignisse, die sie betreffen, zu bearbeiten. Zum Beispiel ist ein Worksheet-Objekt in Excel unter anderem für Mausklicks und Texteingaben in seine Zellen zuständig und auch für das Activate-Ereignis, das eintritt, wenn ein Benutzer oder ein Makro das Arbeitsblatt anwählt und damit aktiviert. Das Worksheet-Objekt reagiert auf diese Ereignisse, indem es eine angeklickte Zelle farblich hervorhebt, eingegebenen Text in einer Zelle anzeigt oder das Arbeitsblatt auf dem Bildschirm sichtbar macht. Für die Tool-Entwicklung ist daran interessant, dass sich VBA-Code in das Standard-Verhalten des Objekts einklinken und dieses erweitern oder modifizieren kann. Dazu werden eigene Ereignisbehandlungs Routinen programmiert. Weiter unten wird gezeigt, wie einfach dies in VBA zu realisieren ist.

Mausklicks, Tastatureingaben und Activate-Ereignisse stammen aus der Systemumgebung oder werden in der Anwendung Excel automatisch ausgelöst. VBA erlaubt darüber hinaus, eigene Ereignistypen zu definieren, die zum Beispiel den bereits genannten Fristablauf anzeigen usw. Dafür sind eigene Klassen zu entwickeln, deren Objekte ein solches Ereignis auslösen (man sagt auch, „ein Ereignis werfen“). Die Objekte dieser Klassen wirken dann als Ereignis-Erzeuger. Eine knappe und klare Erklärung zu selbst definierten Ereignissen bietet etwa [16]. Doch auch eigenentwickelte Ereignisbehandlungsroutinen für Ereignisse, die in Anwendungen bereits standardmäßig vorhanden sind, bieten schon vielseitige Möglichkeiten.

13.5.2 Ereignisbehandlungsroutinen entwickeln

Die VBA-Entwicklungsumgebung unterstützt die Entwicklung von Ereignisbehandlungsroutinen sehr gut. Wenn ein Objekt oder eine Klasse in der Entwicklungsumgebung aktiviert ist, zeigt sie alle Ereignisse an, auf die Objekte der betreffenden Klasse reagieren können. Abb. 13.2 demonstriert dies für die Klasse Document aus dem Objektmodell von Word. Wenn man im VBA-Projektrexplorer das Objekt ThisDocument aktiviert und damit das Code-Modul dieses Objekts öffnet, lässt sich in der linken Auswahlbox am oberen Rand des Code-Fensters seine Klasse Document auswählen. Die rechte Auswahlbox listet anschließend alle Ereignisse auf, die ein Objekt dieser Klasse erkennt. Für andere Objekte im VBA-Projektrexplorer, wie etwa Excel-Arbeitsmappen und Excel-Arbeitsblätter, eigenentwickelte Klassen und auch UserForms und ihre Komponenten funktioniert dies genauso, siehe etwa Abb. 9.3.

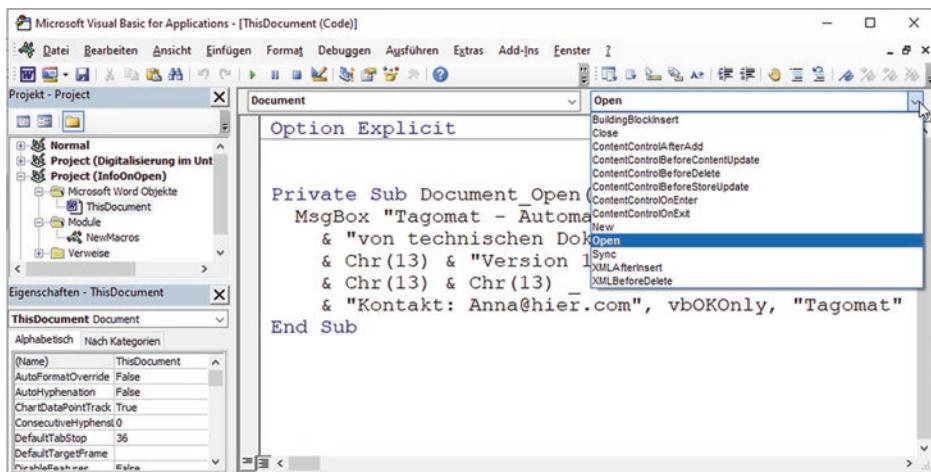


Abb. 13.2 Ereignisprozeduren des Word-Objekts Document

Klickt man nun ein Ereignis in der Auswahlbox an, generiert die Entwicklungs-Umgebung eine Sub mit leerem Rumpf, die man dann zu einer Ereignisbehandlungs-Routine für dieses Ereignis ausbauen kann. Ereignisbehandlungs-Routinen werden in den Modulen angelegt, die zu den betreffenden Objekten oder Klassen gehören. In Abb. 13.2 ist eine Ereignisbehandlungs-Routine für das Open-Ereignis des Word-Dokuments zu sehen. Diese Routine Document_Open, die eine einfache MessageBox anzeigt, wird jetzt bei jedem Öffnen dieses Word-Dokuments automatisch ausgeführt, weil der Prozedurname Document_Open, den die Entwicklungsumgebung generiert hat, die Sub mit dem Open-Ereignis des Document-Objekts verknüpft:

```
Private Sub Document_Open()  
  
    MsgBox "Tagomat - Automatische Verschlagwortung " _  
        & "von technischen Dokumenten " _  
        & Chr(13) & "Version 1.2 (2022)" _  
        & Chr(13) & Chr(13) _  
        & "Kontakt: Anna@hier.com", vbOKOnly, "Tagomat"  
End Sub
```

In Excel und weiteren Office-Anwendungen kann man analog vorgehen, um Makros ereignisbasiert zu starten. Eine Ausnahme bildet PowerPoint, das erst dafür eingerichtet werden muss [17]. In PowerPoint ist es jedoch nicht möglich, beim Öffnen einer Präsentation ohne Eingreifen des Users automatisch ein Makro ausführen zu lassen.

13.5.3 Risiken durch VBA-Makros

Der Rumpf einer Ereignisbehandlungs-Routine kann beliebigen VBA-Code enthalten, auch bösartigen Schad-Code. Word-Dokumente, Excel-Arbeitsmappen und andere Dateien mit Makros, die aus unbekannten Quellen stammen, sind deshalb ein Sicherheitsrisiko. Um dieses zu kontrollieren, sind Makros häufig standardmäßig deaktiviert und können in den Sicherheitseinstellungen gezielt aktiviert werden, etwa für zertifizierte Dateien oder für Dateien, die an bestimmten, als sicher ausgewiesenen Speicherorten abgespeichert sind.

13.6 Ein Makro aus der Symbolleiste für den Schnellzugriff starten

Eine elegante Möglichkeit, um ein Makro starten, besteht darin, es mit einer Schaltfläche in der Symbolleiste für den Schnellzugriff zu verbinden. Die Symbolleiste kann man in Word, Excel, PowerPoint usw. unter **Datei > Optionen > Symbolleiste für den**

Schnellzugriff konfigurieren. Hier lassen sich nicht nur eingebaute Funktionen der Anwendung in die Symbolleiste einfügen, sondern auch selbst programmierte Makros. Diese müssen dazu sichtbar (`Public`) in einem Code-Modul liegen und dürfen keine Argumente benötigen.

Abb. 13.3 zeigt dies für ein Word-Dokument, das ein VBA-Modul `Info` mit einem Makro `InfoZeigen` besitzt. Wenn unter **Befehle auswählen** die Wahlmöglichkeit **Makro** eingestellt ist, erscheint dieses Makro als `Info.InfoZeigen` in der Auswahlliste und kann dort angewählt und der Symbolleiste hinzugefügt werden. Wichtig ist die Einstellung, dass diese Erweiterung der Symbolleiste nicht für alle Dokumente greifen soll, denn in anderen Dokumenten ist das Makro nicht verfügbar. In Abb. 13.3 ist deshalb rechts oben unter der Beschriftung **Symbolleiste für den Schnellzugriff anpassen** das Dokument „`InfoOnOpen.docm`“ ausgewählt. In der Abbildung ist auch die Schaltfläche **Ändern** zu erkennen. Diese öffnet den ebenfalls sichtbaren Popup-Dialog **Schaltfläche „Ändern“**, in dem ein Symbol für das Makro ausgewählt werden kann. Es erscheint dann in der Symbolleiste wie in Abb. 13.4.

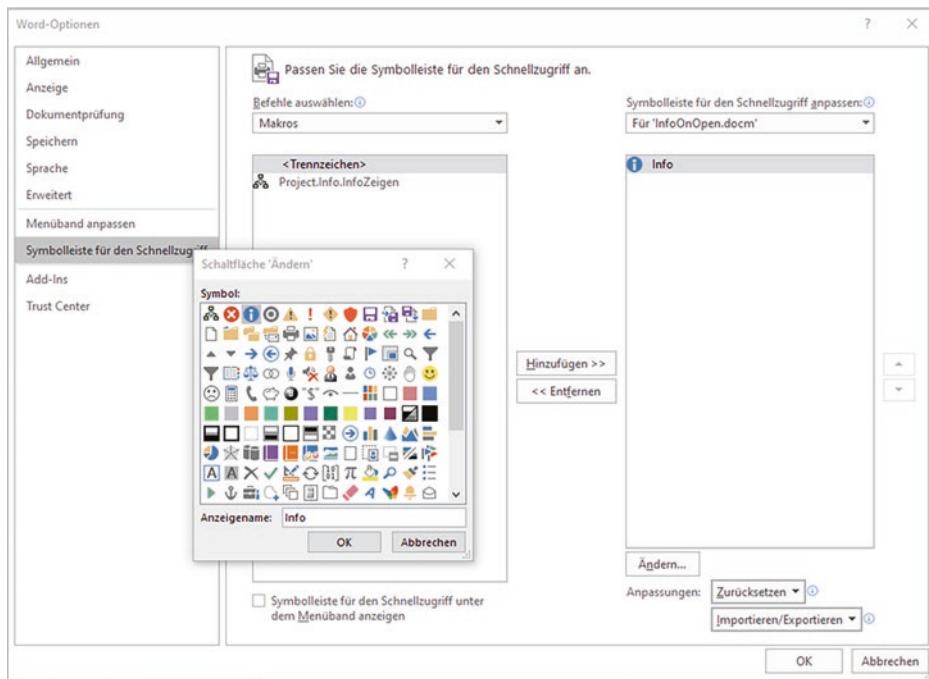


Abb. 13.3 Einfügen eines Makros in die Symbolleiste für den Schnellzugriff

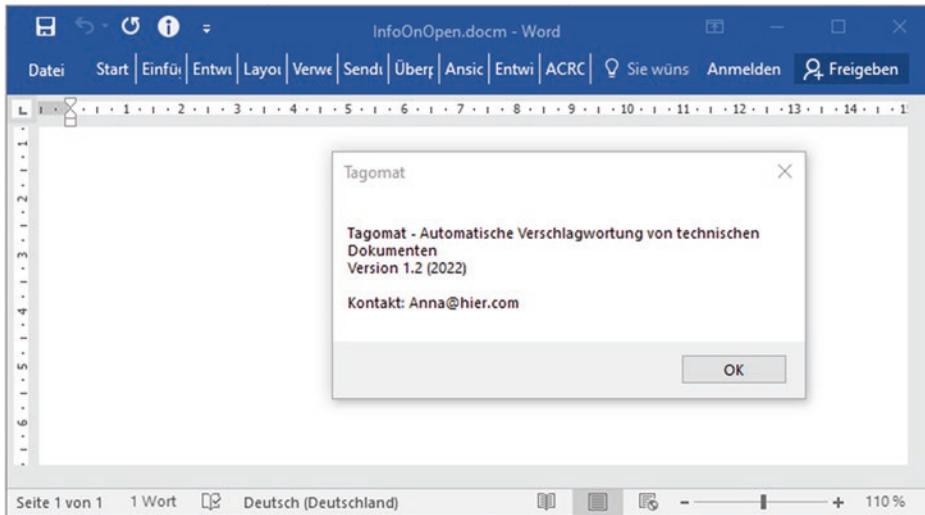


Abb. 13.4 Aufruf eines Makros aus der Symbolleiste für den Schnellzugriff

13.7 Fazit und Ausblick

End User Computing leistet wichtige Beiträge in Unternehmen, insbesondere wenn es gelingt, Schwächen der entwickelten Lösungen zu minimieren. Aktuell findet Low Code-Anwendungsentwicklung in Verbindung mit Robotic Process Automation große Aufmerksamkeit. Low Code-Plattformen sollen auch User ohne tiefgehende Programmierkenntnisse befähigen, Digitalisierungslösungen zu schaffen. Auch hierbei können VBA-Makros integriert werden, um Teilaufgaben zu erledigen [18]–[20].

VBA ist in vielen gebräuchlichen Software-Anwendungen enthalten und somit sofort verfügbar, um kleine und auch größere Digitalisierungslösungen zu erstellen. Weder müssen zusätzliche Entwicklungswerzeuge auf lokalen Arbeitsplatzrechnern installiert noch zentrale Serverlösungen bereitgestellt und administriert werden. Nicht nur die Entwickler, auch die Nutzer sind prinzipiell bereits für die Anwendung von VBA-basierten Tools ausgerüstet, ohne dass zusätzliche Aufwände für Einrichtung oder Betrieb entstehen oder IT-Spezialisten tätig werden müssen. Auch wenn das Budget oder IT-Fachleute knapp sind, bietet VBA die Chance, Ideen in die Realität umzusetzen, schnelle Lösungen zu schaffen und andere, aufwendigere Wege der Digitalisierung vorzubereiten oder zu ergänzen.

Literatur

1. T. Barbe, M. Pezzetti, S. Martin, A. Tovar-Gonzalez, und C. Fluder, „An innovative approach for the design of cryogenic electrical and process control systems at CERN: the cryogenic Continuous Integration project“, *IOP Conf. Ser.: Mater. Sci. Eng.*, Bd. 1240, Nr. 1, S. 012042, Mai 2022, <https://doi.org/10.1088/1757-899X/1240/1/012042>.
2. P. L. Le, T.-M. Dao, und A. Chaabane, „BIM-based framework for temporary facility layout planning in construction site: A hybrid approach“, *CI*, Bd. 19, Nr. 3, S. 424–464, Juli 2019, <https://doi.org/10.1108/CI-06-2018-0052>.
3. P. Santos Bartolome und T. Van Gerven, „A comparative study on Aspen Hysys interconnection methodologies“, *Computers & Chemical Engineering*, Bd. 162, S. 107785, Juni 2022, <https://doi.org/10.1016/j.compchemeng.2022.107785>.
4. T. Okamura *u. a.*, „NMB4.0: development of integrated nuclear fuel cycle simulator from the front to back-end“, *EPJ Nuclear Sci. Technol.*, Bd. 7, S. 19, 2021, <https://doi.org/10.1051/epjn/2021019>.
5. R. Agarwal, P. Dwivedi, R. Mahawar, und A. Karn, „Design and Analysis of Longitudinal Butt Joints Using an Excel/VBA Computational Tool“, in *Recent Advances in Mechanical Engineering*, S. Narendranth, P. G. Mukunda, und U. K. Saha, Hrsg., in Lecture Notes in Mechanical Engineering. Singapore: Springer Nature, 2023, S. 119–130. https://doi.org/10.1007/978-981-19-1388-4_12.
6. U. Feuerriegel, *Verfahrenstechnik mit EXCEL: verfahrenstechnische Berechnungen effektiv durchführen und professionell dokumentieren*. Lehrbuch. Wiesbaden: Springer Vieweg, 2016. <https://doi.org/10.1007/978-3-658-02903-6>.
7. U. Feuerriegel, *Wärmeübertragung mit EXCEL und VBA: wärmetechnische Berechnungen und Simulationen effektiv durchführen und professionell dokumentieren*. Lehrbuch. Wiesbaden [Heidelberg]: Springer Vieweg, 2021.
8. J. Litwin, M. Olech, und A. Szymusik, „Applying Python’s Time Series Forecasting Method in Microsoft Excel – Integration as a Business Process Supporting Tool for Small Enterprises“, *Technical Sciences/University of Warmia and Mazury in Olsztyn*, Bd. 24(1), 2021, <https://doi.org/10.31648/ts.7058>.
9. A. Kopper *u. a.*, „Shadow IT and Business-Managed IT: A Conceptual Framework and Empirical Illustration“, *International Journal of IT/Business Alignment and Governance*, Bd. 9, Nr. 2, S. 53–71, Juli 2018, <https://doi.org/10.4018/IJITBAG.2018070104>.
10. A. Kopper, S. Strahringer, und M. Westner, „Kontrollierte Nutzung von Schatten-IT“, in *IT-GRC-Management – Governance, Risk und Compliance: Grundlagen und Anwendungen*, M. Knoll und S. Strahringer, Hrsg., in Edition HMD. Wiesbaden: Springer Fachmedien, 2017, S. 129–150. https://doi.org/10.1007/978-3-658-20059-6_9.
11. D. Fürstenau, H. Rothe, und M. Sandner, „Leaving the Shadow: A Configurational Approach to Explain Post-identification Outcomes of Shadow IT Systems“, *Bus Inf Syst Eng*, Bd. 63, Nr. 2, S. 97–111, April 2021, <https://doi.org/10.1007/s12599-020-00635-2>.
12. I. Hausladen und P. Sylla, „Business-managed IT – Rahmenbedingungen für mehr IT-Verantwortung durch den Fachbereich“, *HMD*, Juli 2020, <https://doi.org/10.1365/s40702-020-00644-5>.
13. D. Hoogsteen und H. P. Borgman, „Empower the Workforce, Empower the Company? Citizen Development Adoption“, in *HICSS*, 2022, S. 1–10. <https://doi.org/10.24251/HICSS.2022.575>.
14. C. Rentrop, S. Zimmermann, und M. Huber, „Schatten-IT – ein unterschätztes Risiko?“, in *D-A-CH Security 2015 : Bestandsaufnahme – Konzepte – Anwendungen – Perspektiven*, St. Augustin bei Bonn: Syssex, 2015, S. 291–300.

15. N. Urbach und F. Ahlemann, „Schatten-IT als gelebte Praxis – IT-Innovationen werden in interdisziplinären Teams in den Fachabteilungen erarbeitet“, in *IT-Management im Zeitalter der Digitalisierung: Auf dem Weg zur IT-Organisation der Zukunft*, N. Urbach und F. Ahlemann, Hrsg., Berlin, Heidelberg: Springer, 2016, S. 67–75. https://doi.org/10.1007/978-3-662-52832-7_5.
16. Mathieu Guindon, „vba – Is it possible to create and handle a custom Event in a Customized UserForm?“, *Stack Overflow*, 4. April 2018. <https://stackoverflow.com/questions/49660399/is-it-possible-to-create-and-handle-a-custom-event-in-a-customized-userform> (zugegriffen 30. Mai 2023).
17. o365devx, A. Jerabek, K. Brandl, Office GSX, und L. Caputo, „Use Events with the Application Object“, *Microsoft | Learn | Visual Basic for Applications*, 13. September 2021. <https://learn.microsoft.com/en-us/office/vba/powerpoint/how-to/use-events-with-the-application-object> (zugegriffen 29. September 2022).
18. Automation Anywhere, Inc., und Automation 360, „Run macro action“, *AUTOMATION ANYWHERE | Automation 360*, 12. Dezember 2022. <https://docs.automationanywhere.com/bundle/enterprise-v2019/page/enterprise-cloud/topics/aae-client/bot-creator/commands/excel-advanced-package-run-macro-action.html> (zugegriffen 8. Juni 2023).
19. G. Trantzas, A. Deore, T. Maniar, und M. Leon, „Makros in einer Excel-Arbeitsmappe ausführen – Power Automate“, *Microsoft Learn Power Automate*, 16. März 2023. <https://learn.microsoft.com/de-de/power-automate/desktop-flows/how-to/run-macros-excel> (zugegriffen 8. Juni 2023).
20. UIPath, „Aktivitäten – VBA aufrufen“, *UIPath | Dokumentation*, 12. April 2023. <https://docs.uipath.com/de/activities/other/latest/user-guide/invoke-vba> (zugegriffen 8. Juni 2023).