

**Interessante Effekte
mit CSS und SVG** S. 34

**Die neue Version
von NgRx im Detail** S. 58

**Adobe Photoshop
Lightroom CC** S. 114

web & mobile
DEVELOPER

web & mobile **DEVELOPER**

webundmobile.de

Low Code Development statt Programmieren

Wie Sie mit Low-Code-Techniken die Software-Entwicklung dramatisch beschleunigen können S. 18



Aktuelles von Bootstrap

Florence Maurice gibt einen Ausblick auf die künftige Entwicklung des Bootstrap-Frameworks S. 42

Ausgabe 8/19

Deutschland

14,95 EUR

CH: 29,90 CHF

A, B, NL, L:

16,45 EUR



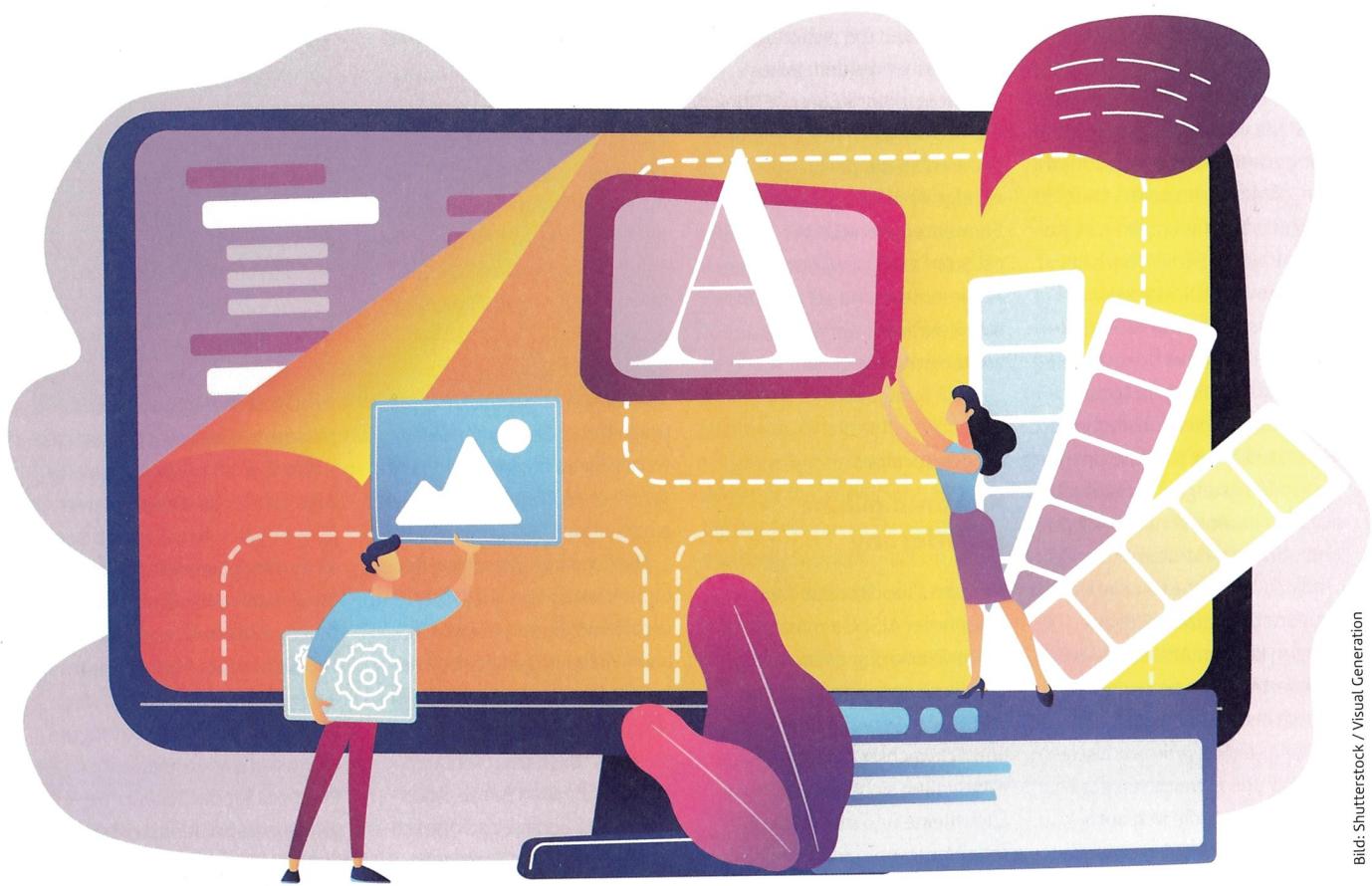


Bild: Shutterstock / Visual Generation

LOW CODE DEVELOPMENT STATT PROGRAMMIEREN

Rasche Entwicklung

Die Softwareentwicklung lässt sich mit verschieden Techniken beschleunigen.

Seit den Urzeiten der Softwareentwicklung sucht man nach Wegen, den Entwicklungsprozess zu beschleunigen und zu vereinfachen. Hierzu wurden in den vergangen Jahrzehnten immer wieder neue Verfahren vorgestellt und auch angewandt. Sie wurden als 4GL, RAD und IDE bezeichnet oder als Service angeboten (PaaS, SaaS, IaaS).

Nun steht mit Low-Code Development (LCD) ein weiterer Versuch ins Haus, die Softwareentwicklung zu beschleunigen. Gebetsmühlenartig werden dabei seit Jahrzehnten die gleichen Argumente hervorgebracht. Low-Code Development soll, wie auch die oben genannten Verfahren der Vergangenheit, eine schnellere Entwicklung ermöglichen. Unternehmen sollen Anwendungen mit geringem Aufwand und minimalem Codieren erstellen können.

Mit einer Low-Code Development Plattform soll der Aufwand für die Programmierung letztendlich auf ein Minimum reduziert werden. Programmierung im herkömmlichen Sinne soll damit weitgehend entfallen. Ferner soll das Verfahren durch die beschleunigte Entwicklung eine rasche Umsetzung der geschäftlichen Anforderungen sowie das schnellstmögliche

Deployment der Anwendungen erlauben. Damit soll es helfen, dem rasanten Wandel in den Geschäftsabläufen gerecht zu werden. Und natürlich soll es auch dem Fachkräfte mangel entgegenzuwirken. Denn Low Code Development soll auch oder vor allem den Mitarbeiter von Fachabteilungen ohne spezielle IT-Ausbildung erlauben, Anwendungen zu erstellen.

Professionellen Softwareentwickler sollen damit nicht notwendig sein. Soweit die Marketingaussagen, die seit Jahrzehnten immer die gleichen sind. Was aber zeichnet die Verfahren wirklich aus? Wie unterscheiden sich die Technologien und welchen Nutzen bringen sie für die Unternehmen tatsächlich? Diese und ähnlichen Fragen wollen wir in dem Beitrag nachgehen ([Bild 1](#)).

Low Code Development – für eine einfache Entwicklung

Zwar gibt es unterschiedliche Implementierungen der LCD-Entwicklungskits, aber im Kern sind sie immer gleich. Durch einen visuellen Editor werden die Benutzerdialoge aufge-

baut. Sie werden meist per Drag & Drop aus einem Katalog zusammengestellt. Eingeschlossen sind ferner Hilfen um den Programmablauf mit Abfrage (if, then, else) zu erstellen. Die so erstellte Anwendung wird dann in einem zentralen Speicher (einem Repository) hinterlegt. Dabei handelt es sich oftmals um einen zentralen Server in der Cloud. Außerdem werden Hilfen für den Test und das Deployment geboten.

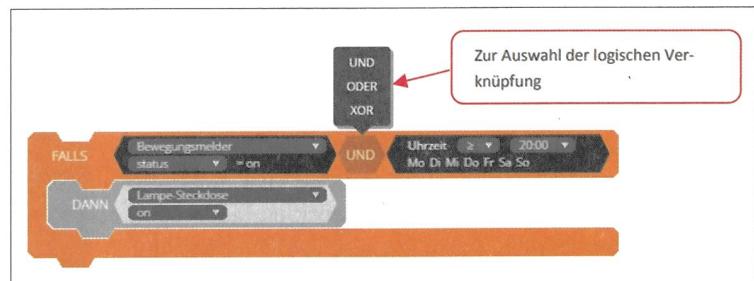
Manche Toolkits bieten ferner Unterstützung für das Projektmanagement, verteiltes Arbeiten, Versionierung und Monitoring. Sind all diese Hilfen aber wirklich neu oder galt dies auch für alle früheren Ansätze wie etwa RAD? Und was ist anders und wie haben sich die RAD-Tools von damals weiterentwickelt ([Bild 2](#)).

Individualprogrammierung vs. Standards

Um diese Fragen zu beantworten hilft ein Blick auf die Anforderungen bei der Softwareentwicklung. Sie war ursprünglich geprägt von der Individualprogrammierung. Doch allmählich kehrte sich der Trend um - weg von der Individualprogrammierung hin zur Ausrichtung auf Standardsoftware.

Die Gründe für dieses Umdenken liegen nicht unbedingt nur in veränderten Notwendigkeiten, sondern haben ihre Ursachen vielmehr in den verringerten Kosten und Möglichkeiten. Beigetragen hat auch der über die Jahre angewachsene Anwendungsstau. Hinzu kommt eine veränderte Arbeitsplatzumgebung. Im Gegensatz zur relative monotonen Welt der Windows-basierten PC findet sich heute eine Vielzahl an unterschiedlichen Geräten parallel im Einsatz.

Neben den traditionellen PC, finden sich heute Notebook, Tablets, Smartphone und eine Armada an individuellen PDAs in den Händen der Nutzer. Ferner ist die Nutzung nicht mehr auf den stationären Einsatz im Büro beschränkt. Die tragbaren Geräte können in der Regel auch ohne festen Netzan-



Durch Designer werden die Anwendungen zusammen geklickt ([Bild 2](#))

schluss benutzt werden, denn andernfalls wären sie oftmals kaum für den Einsatz geeignet. Und auch in Hinblick auf die Softwareentwicklung hat sich viel getan. Während diese ursprünglich fast ausschließlich in den Händen der Entwicklungsabteilung stattfand, trifft man sie mittlerweile auch immer häufiger in den Fachbereichen an. Damit kommen auch Nutzer in den Genuss der Programmierung, die eigentlich von Programmierung nur wenig wissen müssen. Die Grundlage dazu wird durch Standards gebildet.

Werden beispielsweise Bildschirmausgaben durch vorgegebene Standards festgeschrieben, so ist dazu keine individuelle Programmierung mehr notwendig. LCD-Verfahren soll nun helfen diese Individualprogrammierung wo immer es geht zu umgehen. Stattdessen lassen sich gängige Standard-Verfahren und Bibliotheken verwenden.

Vergleich zur Individualprogrammierung

Im Vergleich zur Individualprogrammierung durch die zentrale IT entstehen dabei oftmals preisgünstige und schnelle Lösungen. Damit besteht auch die Möglichkeit zur Programmierung von eigenen Lösungen - und seien dies nur einfachste Makros – durch den Anwender oder die Fachabteilung. Ob das nun ein Segen oder Fluch sein man, hängt vom Einzelfall ab.

Einfache Dinge die schnell benötigt werden und nur wenig Auswirkungen auf die globale IT-Modelle haben lassen sich auch genauso gut in den Fachbereichen erstellen. Hierzu ist es oftmals nicht notwendig, einen Entwicklungsprozess in der zentralen IT anzustossen. Man denke dabei an all die Vielzahl der benutzen Excel-Makros oder ähnlicher Skripte.

Neben ihrem Preisvorteil haben die Produkte von der Stange einen gravierende Vorteil: Sie sind sofort verfügbar. Auf Lösungen durch die zentrale IT-Abteilung müssen oftmals Jahre gewartet werden. Im Gegensatz dazu lassen sich durch manche moderne Entwicklungshilfen Anwendungen viel schneller in Betrieb nehmen. Mit nur wenig Installations- und Konfigurationsaufwand, entstehen dabei maßgeschneiderte Lösungen. Dennoch ist diese Entwicklung nicht ganz ohne Nachteile.

Die Konfiguration auf die Benutzerbelange, entpuppt sich häufig als äußerst mühsam und zeitaufwendig. Die Business-Logik, das Da- ►



Moderne Apps haben mit den Bildschirmmasken der früheren Jahre wenig gemeinsam ([Bild 1](#))

tenbankmodell und die Benutzeroberfläche der Anwendungen werden dabei nicht programmiert, sondern modelliert.

Zwischen Individual- und Standardsoftware?

Durch die Techniken des LCD soll die Entwicklung beschleunigt werden. Aber Verfahren dieser Art sind alles andere als neu. Schon früher gab es ähnliche Ansätze, die auch heute noch angewandt werden. Als Königsweg zwischen dem Einsatz von Standardsoftware und dem herkömmlichen Entwicklungsansatz mit Sprachen der dritten oder vierten Generation waren auch RAD-Systeme propagiert. RAD (Rapid Application Development) zählt zu den früher Vertretenen einer beschleunigten Softwareentwicklung.

Wann kommen allerdings diese zum Tragen und was verbirgt sich hinter dem Schlagwort RAD, LCD, 4GL und all die anderen Verfahren? Und sind die Techniken in den Produkten weit genug ausgereift? Glaubt man den Marketingaussagen der Anbieter, so soll der Entwicklungsprozess damit beschleunigt werden.

Sofern all diese Techniken aber nur eine verkürzte Entwicklungszeit des Applikationscodes ermöglichen, ist deren Nutzen begrenzt und schafft unter Umständen erhöhten Wartungsaufwand in der Zukunft. Oder aber man kauft sich damit Abhängigkeiten von einem Anbieter ein.

Bezieht man die Low-Code-Entwicklung jedoch auf den kompletten Zyklus der Softwareentwicklung, so können sie zu Zeit- und Effizienzgewinn führen. Unter Softwareentwicklung wird in diesem Zusammenhang der vollständige Prozess verstanden. Die auftretenden Tätigkeiten werden im Branchenjargon eingeteilt nach Problemanalyse, Design, Programmierung (Codierung), Tests (Qualitätssicherung) mit Fehlersuche, Einweisung der Benutzer, Dokumentation, Software- und Konfigurationsmanagement und schließlich der nachfolgenden Wartung beziehungsweise Weiterentwicklung (Change Management) oder Pflege.

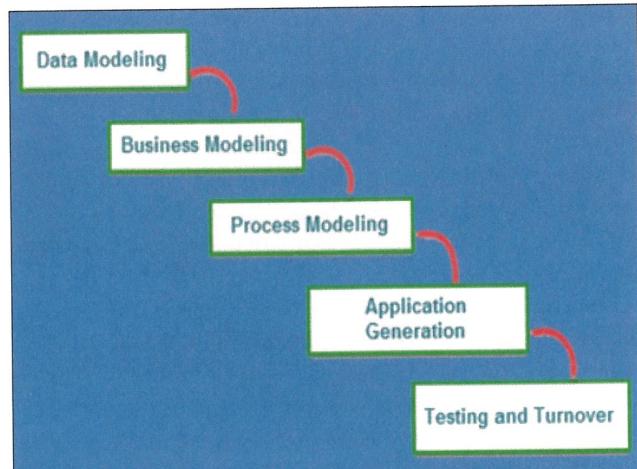
Begleitet werden die einzelnen Schritte von ihren Ergebnissen, wie zum Beispiel Pflichtenheft, Lastenheft, Datenmodell, Funktionsmodell, Prozess Modell, Applikationscode, Programm- oder Schulungsdokumentation (**Bild 3**).

Das Wasserfallmodell als Flaschenhals

Das klassische Vorgehen in der Softwareentwicklung wird häufig als Produktion nach dem Wasserfallmodell bezeichnet. Ein Kriterium dieses Verfahrens ist seine streng sequentielle Abarbeitung.

Ein nachfolgender Entwicklungsschritt wird erst nach dem Abschluss der vorausgehenden Aktivitäten eingeleitet. Das Ergebnis einer Stufe wird als Basis für den nachfolgenden Schritt verwendet. Diese Ausrichtung am Taylorismus ist das Fundament für den Großteil der Abläufe in unserer heutigen Arbeitswelt.

Hierbei wird ein Bearbeitungsvorgang in viele kleine Schritte zerlegt, die seriell oder parallel, durch die Verarbeitungskette geschleust werden. Seit Beginn des 20. Jahrhunderts beherrschte diese Vorgehensweise unsere Arbeitswelt. Was Jahrzehnte gut war kann so falsch nicht sein. Oder doch? Die verstärkte Orientierung am Kunden, häufige Änderun-



Durch die Modellierung der Anwendung soll deren Entwicklung beschleunigt werden (**Bild 3**)

gen der gesetzlichen oder wirtschaftlichen Rahmenbedingungen erfordern von den Unternehmen eine schnelle Ausrichtung der Geschäftsabläufe an den Vorgaben.

Cloud, Industrie 4.0, IoT, 5G sind einige der aktuellen Treiber dieser Entwicklung. Aber deren Dynamik lässt sich immer schwieriger mit der schrittweisen Abarbeitung der Entwicklungsprozesse umsetzen. Stellen sich nämlich bei nachgeschalteten Entwicklungsschritten Probleme mit den Ergebnissen früherer Stufen ein, so sind mitunter alle bisherigen alle Schritte neu zu durchlaufen.

So kann sich beispielsweise bei der Programmierung herausstellen, dass durch geringfügige Änderung des Daten- oder Objektmodells ein weitaus einfacheres und wartungsfreundlicheres Programm entsteht. Ist nun das Programm anzupassen oder sollte das Datenmodell überarbeitet werden?

Im Zweifelsfall wird der zweite Weg eingeschlagen. Oder aber der Programmierer optimiert sein Programm ohne Rücksicht auf die Modelle. Auch die parallele Bearbeitung von Einzelschritten wird durch den streng sequentiellen Ablauf erschwert. Dies wäre allerdings eine notwendige Voraussetzung, um eine geringere Entwicklungszeit für die gesamte Applikation zu erhalten. Das Ergebnis sind Durchlaufzeiten bis zu mehreren Jahren.

In der Folge zeigt sich immer häufiger, dass die ursprüngliche Definition, welche ja in Abstimmung mit dem Benutzer getroffen wurde, bis zum Zeitpunkt der Realisierung in Code und Bildschirmmasken nicht mehr zutrifft. Die Engpässe des Wasserfallmodells führen dazu, dass die Unternehmen sich verstärkt nach moderneren und vor allem schnelleren Varianten wie eben LCD oder RAD umschauen.

In vielen Unternehmen ist außerdem eine weitere Entwicklung zu beobachten: Die Fachbereiche beginnen verstärkt, die Entwicklung von Geschäftsanwendungen in eigene Hände zu nehmen oder dies zumindest personell zu begleiten. Auch diese Tendenz beruht im Wesentlichen auf den bereits erwähnten Ursachen: Dem Anwendungsstau und den komplexer und in größerer Frequenz auftretenden Änderungen der Geschäftsregeln.

Mitunter erscheint es einfacher, einem Sachbearbeiter Excel beizubringen, als einen Programmierer in das Steuerrecht einzuweisen. Übersehen wird dabei allerdings, dass durch individuelle Schnellschüsse die Altlasten von morgen produziert werden. Um möglichen Fehlentwicklungen vorzubeugen, müssen die IT-Fachkräfte aktiv auf ihre Nutzer zugehen und in den Dialog über die Anforderungen und Realisierungen der Software eintreten (**Bild 4**).

Interaktion und Iteration

Um den Anwendungsstau in der Softwareentwicklung abzubauen und gleichzeitig die Akzeptanz der zentralen IT in den Unternehmen zu fördern, muss eine verstärkte Ausrichtung an den Belangen des Kunden, verbunden mit einer beschleunigten Entwicklung erfolgen. Die zentralen Anforderungen an die Softwareentwicklung ist daher Interaktivität. Im Gegensatz zum streng sequentiellen Ablauf erfolgt nun die Entwicklung unter Einsatz von schnellen Techniken einem Kreislauf, der immer wieder durchlaufen wird, bis schließlich der vorläufige Zielpunkt erreicht ist.

Ausgehend von den ersten Ergebnissen der Analyse werden relativ bald die weiteren Entwicklungsschritte durchlaufen. Beim Auftreten von Fehlern oder Engpässen wird der Vorgang wiederholt. Die Teilergebnisse sind jeweils mit den Benutzern (Kunden) abzustimmen. Mängel und Korrekturen werden dabei frühzeitig erkannt. Gleches gilt, wenn beispielsweise noch Informationen fehlen oder Voraussetzungen zu klären sind. Auch paralleles Arbeiten an den verschiedenen Entwicklungsstufen ist machbar.

Dennoch soll dieser Ansatz nicht dazu führen, ohne ein sinnvolles Konzept und ohne Planung an die Entwicklung heranzugehen. So einfach und schnell durch die modernen Werkzeuge einen Dialog mit dem Benutzer zu gestalten ist, die Frage des vorausschauenden Designs der Geschäftslogik wird damit nicht einfacher. Die Herausforderung für die Entwicklung besteht darin, ein möglichst abstraktes Vorgehen zu finden, wobei die prinzipiellen Abläufe gleich bleiben sollen, die konkrete Realisierung mit Datenfeldern, Aktionen etc. sich aber ändern darf.

Das Vorgehen erfordert allerdings eine Voraussetzung, welche noch nicht hinreichend unterstützt wird: Die Tools müssen gut aufeinander abgestimmt sein und neben Forward- auch Reverse-Engineering unterstützen. Ansonsten gehen die Änderungen, welche in einem Schritt gemacht worden sind, beim nächsten Durchlauf verloren. Bezieht man sich auf obige Einteilung, so ergeben sich durchaus Ansätze und Hilfestellung für eine beschleunigte Softwareentwicklung.

Analyse- und Designphase

Die Analyse des Anwendungs- oder Kundenproblems kann durch den Einsatz von Tools vereinfacht und beschleunigt werden. Viele der gebotenen Werkzeuge sind mittlerweile in der Lage, aus dem Anwendungsdesign Skelette von Programmen oder Skripte für die Interaktion mit den Daten abzuleiten. Das darf jedoch keinesfalls nur in eine Richtung erfolgen. Für die gewaltigen Mengen an bestehenden Datenbeständen beziehungsweise Anwendungen soll der Weg des Reverse Engineering nicht verbaut sein.

Somit ist die Möglichkeit offen, an beiden Stellen, der Modelldarstellung als auch Implementierung, zu verändern und zu optimieren. Auch für die Entwicklung nach objektorientierten Methoden gibt es eine Vielzahl von Werkzeugen für den Analyse- und Designprozess.

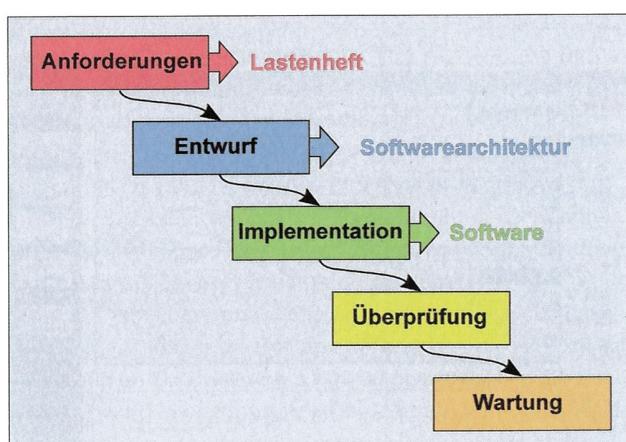
Masken und Benutzerinterface

Die zentrale Schnittstelle zwischen den Implementierungsdetails des Programms und den Anforderungen der Benutzer repräsentiert sich in den Bildschirm- oder App-Dialogen. Für deren Generierung bietet der Markt mittlerweile eine enorme Auswahl an Werkzeugen. Durch den Einsatz von grafischen Entwicklungs-Tools kann das Design und der Ablauf der Dialogmasken direkt zusammen mit dem Anwender erstellt werden.

Bei diesem Teil der Entwicklung kann der Anwender frühzeitig teilhaben und seine Anforderungen einbringen. Das separate Zeichnen von Benutzbildschirmen, welche später in Code umgesetzt werden, reduziert sich auf ein Minimum. Sollen Anwendungen nach den traditionellen Entwicklungs-schemata für verschiedene Grafikmanager erstellt werden, so ist darauf zu achten, dass das Tool die Anbindung an die benötigten Dialog-Implementierungen unterstützt. Soll als Zielumgebung hingegen bereits ein Web-Browser mit HTML-Interpretation eingesetzt werden, so existieren auch hierfür bereits verschiedene Hilfen.

Um die Erstellung der Masken und Bildschirme zu vereinfachen, setzt man seit Jahren auf Drag & Drop bei der Entwicklung. Dabei werden die benötigten Interaktionselemente aus einem Baukasten auf die zu erstellende Maske gezogen. Anschließend werden diese Aktionselemente mit dem notwendigen Code im Hintergrund verknüpft. Hilfreich sind dabei auch zusätzliche Designelemente wie etwa Flussdiagramme und Tabellen. Sie können helfen, den Entwicklungs-aufwand weiter zu reduzieren.

Bezogen auf die eigentliche Implementierungssprache kann festgestellt werden, dass LCD nicht an eine Program- ►



Wasserfall: Bei der Softwareentwicklung nach dem Wasserfall-Modell erfolgt die Entwicklung sequentiell (**Bild 4**)

miersprache gebunden ist. Beim ausschließlichen Einsatz von gängigen Programmiersprachen der dritten Generation wie C, C++ oder anderen Sprachen sind die Möglichkeiten für LCD-Techniken allerdings relativ eng begrenzt.

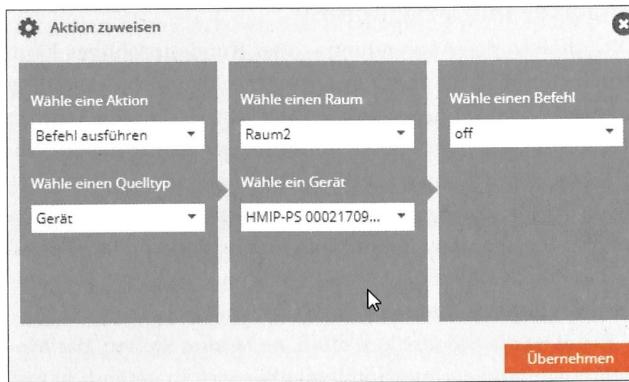
Aber auch deren Nachfolger wie etwa Java, Python oder all die Skriptsprachen bieten hier schon mehr Unterstützung. Sie nehmen häufig einen Großteil der mühevollen Kleinarbeit ab und erlauben es, sich mit dem eigentlichen Applikationspro-

sich für traditionelle Entwicklungssprache mit dem Mehraufwand heute und einfacheren Portierungsmöglichkeiten in der Zukunft oder es wird ungeachtet einer gesicherten Weiterentwicklung des Werkzeugs mit moderneren Tools entwickelt. Der dabei erzielte Zeitgewinn mag später in einer notwendigen Portierung wieder verloren gehen.

Der beste Weg wäre die Nutzung eines Entwicklungstools der vierten Generation, das wiederum als Codegenerator für schnellere Umgebungen (zum Beispiel C, C++, Java, Python) dient. Doch das ist nicht immer gegeben. Die Hersteller wollen die Kunden an sich und die eigenen Werkzeuge binden und auch in Zukunft Lizenzen verkaufen. Dennoch gibt es auch Ausnahmen.

Vor allem Java bot für manchen Hersteller von Entwicklungssoftware wieder eine Chance, um in den Markt einzusteigen oder Terrain gutzumachen. Aber auch aus Performancegründen ist dieser Ansatz interessant.

Für einen schnellen Prototyp ist das Interpretieren des Programmcodes oder die Erzeugung von P-Code oder Byte-Code sinnvoll. Für die endgültige Programmversion offeriert die Übersetzung in Maschinencode (beziehungsweise C-Code) Laufzeitverbesserungen. Auch in der Java-Umgebung wird dieser Weg beschritten - zugegebenermaßen mit etwas anderem Schwerpunkt (Portabilität, Laufzeitverhalten).



Die Aktionen stoßen Verarbeitungen an (Bild 5)

blem auseinanderzusetzen. Sie umfassen oftmals Hilfen für das Design der Benutzerdialoge, dem Debugging und weiteren Aufgaben, die sich im Zuge einer Applikationsentwicklung ergeben. Gleichzeitig weisen sie im Vergleich zu den oben erwähnten Drittgenerationsprachen Laufzeitnachteile und den proprietären Charakter auf (Bild 5).

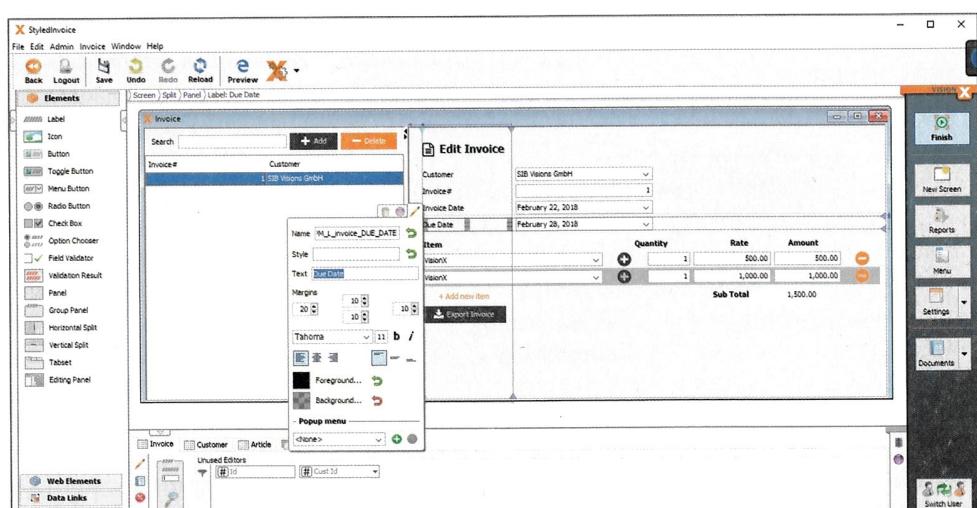
Letzteres bedeutet, dass die Entscheidung für ein LCD-Werkzeug immer auch eine Vertrauensfrage in die Marktmacht und Zukunft des Herstellers sein wird. Denn wenn nicht gewährleistet ist, dass das Werkzeug auch in einigen Jahren noch unterstützt wird oder zumindest eine Portierung auf andere Systeme möglich ist, behindert dies jegliche Weiterentwicklung der damit erstellten Anwendungen.

Allerdings darf dabei aber nicht verschwiegen werden, dass auch eine Entscheidung für eine Entwicklungssprache der dritten Generation immer von den Basisbibliotheken oder Objektklassen des Toolherstellers abhängt. Dies gilt heute wie damals. Wenn niemand mehr da ist, der das Framework (ob C, Python, JavaScript oder HTML) weiterentwickelt, veraltet es und damit all die mit ihm erstellten Anwendungen. Als Entscheidungshilfe pro oder contra LCD, RAD oder 4 GL zeigt sich daher: Entweder man entschließt

Berichte und Auswertungen

Für jegliche Auswertungen oder Listen werden in der Regel schon seit Jahren Reportgeneratoren herangezogen. Tools dieser Kategorie übernehmen einen Großteil der Programmierung, sie sind also längst in die Gruppe der Low Code Development-Tools einzugruppieren.

Um sie universeller einsetzen zu können, sollten diese jedoch über eine umfangreiche Programmiersprache sowie ein leistungsfähiges und graphisches Designwerkzeug verfügen. Um eine einfache Einbindung in das gesamte Anwendungssystem zu erreichen, sollte das Werkzeug in jedem Fall über diverse Programmier- und Kommunikationsschnittstellen verfügen. Über Mechanismen dieser Art sind die Produkte



Durch GUI-Designer werden die Oberflächen erstellt (Bild 6)

von außen zu steuern und mit Parametern zu versieren. Auch automatische Abläufe ohne Userinteraktionen – sind dadurch erzielbar (Bild 6).

Bibliotheken

Bibliotheken gehören zu jeder Entwicklungsumgebung zwingend dazu. Sie werden meist als reine Funktionsbibliotheken oder Klassenbibliotheken angeboten. Letztere sind in der Struktur ihrer Anordnung und Hierarchie vorteilhafter.

Durch die Vererbung über mehrere Stufen sind aus relativ kleinen Klassen umfangreiche Konstrukte mit abgegrenztem Aufgabenumfang aufzubauen. So wie aus der strukturierten Verknüpfung von Feldern schließlich Datensätze und Datenbanken werden, wird aus einer Vielzahl von Feldern mit Ein-/Auszabelogik schließlich der Geschäftsablauf Erfasse Kundenstamm. Das Objekt Kunde und dessen Geschäftsprozess lässt sich später als eine Einheit behandeln und ansprechen - und als solche in weitere Anwendung integrieren.

Weitere Geschäftsobjekt in diesem Kontext sind beispielsweise ein Auftrag beziehungsweise der Prozess Erfasse Auftrag oder Erstelle Lieferschein. Im Idealfall sind die konkreten Ausprägungen des Objektes Kunde bei Bedarf durch die Menge und Spezifikation der benötigten Felder zu parametrisieren.

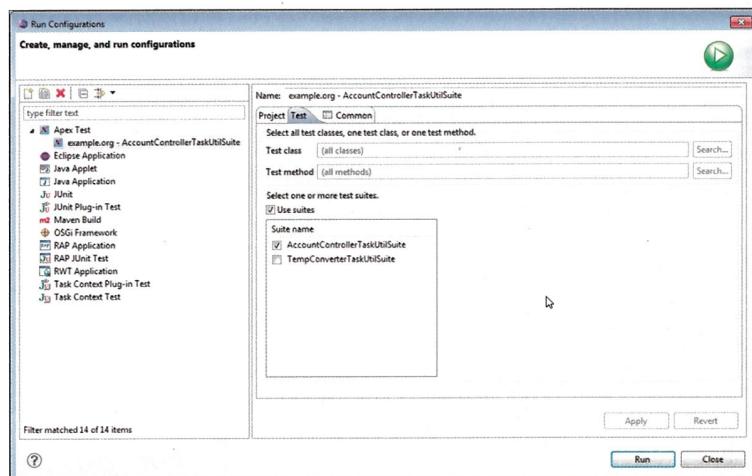
Programmentwicklung durch Frameworks und Templates

Führt man den Gedanken fort, so entstehen aus der Komposition vorhandener Klassen, angereichert um eigene Logik, Rahmenwerke (Frameworks) für das Anwendungsdesign. Auf die generischen Gebilde der Frameworks wird bei späteren Entwicklungsschritten zurückgegriffen.

Derzeit stellen Frameworks einen vielversprechenden Weg der effizienten Softwareentwicklung dar. Dabei muss allerdings bedacht werden, dass das Framework als solches erstmal geschaffen werden muss. Somit ist Voraarbeit zu leisten, die sich erst in späteren Entwicklungsprojekten positiv auswirken wird. Frameworks werden für unterschiedlichste Zwecke aufgebaut. Die Applikations-Frameworks stellen ein Grundgerüst dar, die einen Aufbau von Anwendung durch weitgehenden Rückgriff auf die Framework-Bausteine und Anwendungskomponenten gestatten.

Application Enabling-Frameworks hingegen haben unterstützenden Charakter (zum Beispiel Netzanbindung, Benutzerdialog, Sicherheit etc.) für die Anwendungen. Gegenüber reinen Klassen- oder Funktionssammlungen wird bei Frameworks auch die Wiederverwendung gesteigert.

Statt auf einzelne Funktionen oder Klassen zurückzugreifen und diese immer wieder neu zu gruppieren, wird diese Anordnung im Framework nur einmal getroffen. Um die Entwicklung von Anwendungen zu beschleunigen, greift man gerne auf Templates zurück. Templates haben sich in der IT in vielen Bereich etabliert. Application-Templates helfen beim Bau von Anwendungssystemen. Ein Template ist immer



Moderne Entwicklungssysteme unterstützen mehrere Standards (Bild 7)

eine Vorlage für eine Anwendung oder auch ein anderes System. Durch deren Nutzen beschleunigt sich der Aufbau. Die Idee, auf der Basis von Templates und Frameworks Anwendungen zu erstellen klingt bestechend und macht viele Anleihen bei der objektorientierten Programmierung. Diese hat sich bekanntlich die Wiederverwendung von Ergebnissen früherer Entwicklungsschritte auf die Fahnen geschrieben. Erzielt wird diese Re-Usability durch die Vererbung in den Klassen beziehungsweise durch die Verwendung von Vorlagen (Templates).

Trotz der Verlockung der komponentenbasierenden Entwicklungsmethodik sind die grundlegenden Techniken weder neu noch trivial in der Umsetzung. Neu sind sie deswegen nicht, weil in der Softwareentwicklung das Prinzip der Wiederverwendung ohnehin immer schon vorherrschte. Der Gebrauch von Unterprogrammen, Makros oder Bibliotheken entspricht per Definition der Wiederverwendung von Code und dessen innenwohnender Logik.

Auch die Funktionsbibliothek oder das Runtimesystem von Sprachen beruhen auf den gleichen Forderungen. Die Herausforderung dabei aber ist auch durch die Kapselung von Anwendungsteilen ein klares und strukturiertes Vorgehen. In den folgenden Erläuterungen wollen wir beispielhaft auf einige gängige LCD-Werkzeuge eingehen, um den typischen Ablauf und Aufwand aufzuzeigen. Die Aufstellung ist aber nicht als Wertung zu verstehen. Es gibt darüber hinaus auch weiter vergleichbare Toolsets (Bild 7).

Low Code Development für Smart Home-Apps

Als Beispiel für eine recht aktuelle Version eines Entwicklungskits für Smartphone-Apps wollen wir den AIO Creator Neo herausgreifen. Das Entwicklungswerkzeug hilft bei der Erstellung von Apps für Smart Home-Anwendungen. Der Hersteller spricht dabei von einer Fernbedienung. Dies wird vollständig mit dem Entwicklungskit erzeugt und verwaltet – ganz ohne Programmierung im herkömmlichen Sinn.

Beim Anlegen der Fernbedienung ist das Zielgerät auszuwählen und zu spezifizieren. Dies ist die Stelle, an der beispielsweise festgelegt wird, ob ein Apple iPhone, ein ►

Samsung Galaxy oder ein beliebiges anderes Smartphone oder Tablet zum Einsatz kommen soll. Die Darstellung auf dem Bildschirm wird dabei automatisch an das Zielgerät angepasst. An dieser Stelle zeigt sich die weitere oben erwähnte Geräteunabhängigkeit dieses Toolset. Eine App wird in der Regel immer aus mehreren Seiten bestehen. Im Hauptfenster in der Mitte werden die einzelnen Seiten der App dargestellt. Und rechts im Bild sind schließlich die Eigenschaften des jeweils gewählten Objektes eingeblendet. Unsere App soll unterschiedliche Geräte steuern.

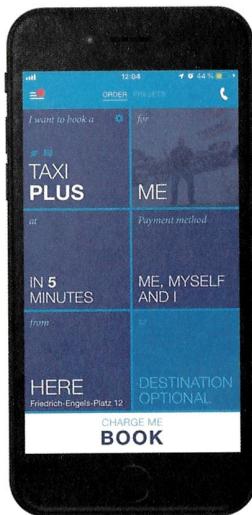
Diese werden durch den Geräte-Manager verwaltet. Unterstützt durch die Menüs und Hilfen erfolgt nun der seitenweise Aufbau der gesamten Fernbedienung. Die Geräte werden dabei per Drag & Drop aus den Menüs auf die Seiten gezogen. Für die wichtigsten Bedienobjekte liefert das Entwicklungswerkzeug eine Vielzahl an vordefinierten Objekten und Icons. Icons können in der Größe verändert, gespiegelt oder rotiert werden, ganz ohne Programmierung.

Auch die Darstellungsebene und Position sind wählbar. Ferner lässt sich der jeweilige Hintergrund individuell anpassen. Über Buttons werden die Aktionen angestoßen. Auch diese Schritte werden durch Drag & Drop zusammengestellt, ganz ohne Programmierung in einer herkömmlichen Sprache. Für komplexere Prozesse stehen Makros zur Verfügung.

Für Java und Eclipse: VisionX

VisionX wirbt auf seiner Website mit einer wesentlich schnelleren Entwicklungszeit gegenüber der traditionellen Programmierung. Das Entwicklungskit basiert vollständig auf Java Open Source-Frameworks. Damit sind die Anwendungen unabhängig von proprietärer Software. VisionX erzeugt Java Source Code. Dieser Source Code kann später, wenn gewünscht, weiter verfeinert und angepasst werden. Die weitere Bearbeitung kann beispielsweise in Eclipse erfolgen. Direkt im Code lassen sich auch Dinge implementieren, die über den Designer nicht direkt erzeugt werden können.

VisionX erlaubt auch ein Reverse Engineering, das heißt, der veränderte und angepasste Java-Code kann anschließend im Designer wieder überarbeitet werden. Die eigenen Anpassungen werden dabei nicht berührt und bleiben als solche erhalten. Der Aufbau einer Anwendung folgt dabei dem im Text erwähnten Schema. Beim Design einer neuen Applikation greift der Entwickler auf ein bestehendes Template zurück. Dieses kann nach Belieben geändert und angepasst werden. Die Masken stellen das Benutzerinterface dar. Im Falle von Anwendungen für mobile Geräte sind es die App-Bildschirme. Anschließend müssen die Masken mit den Daten verknüpft werden. Hierzu stehen Konnektoren bereit. Als Datenquellen kommen bestehende Datenbanken oder auch einfache Datenquellen wie etwa ein Excel-Arbeitsblatt in Betracht. Es lassen sich aber auch gänzlich neue Datenmodelle aufbauen und anbinden. Das dritte Glied einer Applikation



Eine mit LCD erstellte Anwendungs-App (Bild 8)

ist immer die eigentliche Codelogik. Hierbei wird oftmals auch von der Businesslogik gesprochen. Sie kann in VisionX auf drei Arten erzeugt werden. Entweder greift man auf einen Workflow-Designer zurück oder man nutzt den internen Datenfluss. Die dritte Variante ist, den Code selbst in Java zu erzeugen (Bild 8).

Templates: Infanywhere

Ein weiteres Werkzeug das wir vorstellen wollen, stammt von Infanywhere. Auch damit verläuft die Anwendungsentwicklung ähnlich wie bei den erwähnten Toolsets. Im ersten Schritt muss sich der Benutzer auf der Webseite des Anbieters registrieren. Dann kann er auch schon mit einer neuen Applikation loslegen. Auch Infanywhere nutzt Templates. Auf der Webseite werden insgesamt acht unterschiedliche Templates angeboten. Dies sind unter anderem Templates für eine digitale Bibliothek, ein

Kontaktverzeichnis, einen Produktkatalog, Events und Ereignisse, ein Event-Kalender und für eine Personalverwaltung. Mithilfe dieser unterschiedlichen Templates lassen sich recht umfangreich Anwendungen aufbauen. Daneben steht aber auch die Möglichkeit bereit, eine Anwendung völlig frei zu designen. Anschließend erfolgt der Aufbau der Masken im Designer. Hierbei ist auch die Navigationsfolge der Anwendung zu bestimmen (Bild 9).

Fazit

In der Zusammenfassung lässt sich feststellen, dass durch den gezielten Einsatz geeigneter Werkzeuge die Entwicklungszeiten durchaus reduziert werden können. Für schnell zu er-

Vorbereitet Templates vereinfachen die Anwendungs-Entwicklung in Unternehmen (Bild 9)

Links zum Thema

- AIO Creator Neo
<https://www.mediola.com/ccu3/>
- VisionX
<https://visionx.sibvisions.com>
- Infanywhere
<https://infanywhere.com>

stellende Anwendungen die vielleicht auch nur vorübergehende benötigt werden, ist dies bestimmt ein brauchbarer Ansatz. Soll allerdings die erzeugte Anwendung auch langfristig im Einsatz sein und an bestehende Daten und Geschäftsprozesse angebunden werden, so muss darauf geachtet werden, dass die Anwendung auch in Zukunft gewartet werden kann.

Das optimale Werkzeug ist offen, unterstützt Standards und erlaubt interaktives und iteratives Vorgehen. Je näher man dem Ideal ist, umso weniger Aufwand, bezogen auf den vollständigen Lebenszyklus der Anwendung, wird benötigt. Low-Code Development Plattformen können recht umfangreich werden. Keineswegs sollte man sich der Illusion hingeben, dass man mit Low Code-Hilfen gänzlich ohne Programmier-Knowhow auskommt. Auch wenn man mit den Werkzeugen keine Sprache wie Java oder dergleichen erlernen muss, sollte man doch in der Lage sein, Geschäftslogik in Prozesse umzusetzen.

Ferner müssen mitunter Datenmodelle verstanden, angepasst und geändert werden. Außerdem wird Wissen verlangt, um Entwicklung, Test, Versionierung, Build und Deployment erfolgreich durchzuführen. Durch Low-Code Development Plattformen werden diese Schritte minimiert und weitgehend automatisiert. Dies kann die Entwickler erheblich entlasten. Sie können sich stattdessen mehr auf die eigentliche Geschäftslogik konzentrieren. Durch die vorbereiteten Templates wird auch der Test von Anwendungen einfacher oder fällt gänzlich weg.

Gegenüber einer textuellen Programmierung, egal in welche Sprache, wird damit der ganze Aufbau zwar mehr bebildert aber auch umfangreicher. Als Entwickler mit Erfahrung in vielerlei Programmiersprachen bevorzuge ich den traditionellen Weg. Jemand der noch nie mit einer Entwicklungssprache zu tun hatte, wird sich beim grafischen Pendant vielleicht eher zuhause fühlen.



Johann Baumeister

hat Informatik studiert und langjährige Erfahrung in der Entwicklung, Anwendung und dem Rollout von IT-Systemen. Außerdem ist er als Autor für zahlreiche IT-Publikationen tätig.

Lernen, Verstehen, Anwenden

Angular für Fortgeschrittene

Trainer: Gregor Woiwode
3 Tage, Köln/Karlsruhe



Progressive Web App Bootcamp

Trainer: Peter Kröner
3 Tage, Köln/Berlin



Einstieg in PHP

Trainer: Alexander Hofmann
2 Tage, Köln/München



Effiziente Softwarearchitekturen mit PHP

Trainer: Stefan Priebsch
2 Tage, München



Versionsverwaltung mit Git

Trainer: Marco Beelmann
2 Tage, Köln/Stuttgart



Einstieg in Python

Trainer: Mike Bild
3 Tage, Köln



App-Entwicklung mit Xamarin

Trainer: Sebastian Seidel
3 Tage, Köln



••• Termine nach Absprache •••

Weitere Informationen unter
developer-media.de

CONTENTS

- 2.1. Overview
- 2.2. Processing Model
- 2.2.1. Features
- 2.2.2. Dynamic interactive mode
- 2.2.3. Animation mode
- 2.2.4. Selection mode
- 2.2.5. State
- 2.2.6. Secure state
- 2.3. Processing model for resource documents
- 2.3.1. Examples
- 2.4. Document Conformance Classes
- 2.4.1. Conforming SVG DOM Subtrees
- 2.4.2. Conforming XML Fragments
- 2.4.3. Conforming XML-Compatible SVG Markup Fragments
- 2.4.4. Conforming XML-Compatible SVG DOM Subtrees
- 2.4.5. Conforming SVG Stand-Alone Files
- 2.4.6. Error processing
- 2.5. Software Conformance Classes
- 2.5.1. Conforming SVG Generators
- 2.5.2. Conforming SVG Authoring Tools
- 2.5.3. Conforming SVG Servers
- 2.5.4. Conforming SVG Interpreters
- 2.5.5. Conforming SVG Viewers
- 2.5.5.1. Printing implementation notes
- 2.5.6. Conforming High-Quality SVG Viewer

• is rooted by an '`svg`' element in the SVG namespace,

• conforms to the content model and attributes rules for the elements defined in this document (Scalable Vector Graphics (SVG) Specification), and

• conforms to the content model and attributes rules defined by other specifications for any elements in the SVG namespace defined by those specifications, including [filter-effects-1], [css-masking-1], [svg-animation-1], and [scripting-1].

SVG documents can be included within parent XML documents using the XML namespace facilities described in [Namespaces in XML 2.0]. Note, however, that since a conforming SVG DOM subtree has an 'svg' element as its root, it cannot be included in an XML document as an element, because XML does not allow roots. Thus, the SVG document must be included as an entire document, using the use of an individual non-'`svg`' element from the SVG namespace is not allowed.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE SomeParentXMLGrammar PUBLIC "-//SomeParent//DTD SomeParent//Grammar.dtd">
<SomeParentXML>
  <!-- Elements from ParentXML go here -->
  <!-- The following is not conforming -->
  <rect xmlns:z="http://www.w3.org/2000/svg"
        x="0" y="0" width="10" height="10" />
  <!-- More elements from ParentXML go here -->
</SomeParentXML>
```

Instead, if you want to include a conforming SVG document tree, the file must be modifiable as follows:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE SomeParentXMLGrammar PUBLIC "-//SomeParent//DTD SomeParent//Grammar.dtd">
<SomeParentXML>
  <!-- Elements from ParentXML go here -->
  <!-- The following is conforming -->
  <svg xmlns:z="http://www.w3.org/2000/svg"
       width="100px" height="100px">
    <z:rect x="0" y="0" width="10" height="10"/>
  </svg>
  <!-- More elements from ParentXML go here -->
</SomeParentXML>
```

The SVG language and these conformance criteria provide no designated size limits on any aspect of SVG content. There are no maximum values on the number of elements, the amount of character data, or the number of characters in attribute values.

CSS UND SVG

Interessante Effekte

Mit der Kombination von CSS und SVG lassen sich interessante Effekte erzielen.

Skalierbare Vektor Graphiken (SVG) haben eine lange Historie. Schon im Jahr 2001 veröffentlichte die W3C die erste Spezifikation. Seit dem Oktober 2018 gibt es eine Candidate Recommendation für die Version 2.0. In der Zwischenzeit hat sich einiges getan und alle aktuellen Browser unterstützen einen Großteil des Sprachumfangs.

In diesem Workshop zeigen wir Ihnen, welche Effekte Sie mit den Filter-Effekten von SVG erzielen können. Dabei richten wir den Fokus zum einen auf Objekte, die mit SVG erzeugt werden und zum anderen auf Inhalte einer Webseite.

Das Filter-Element

Viele von Ihnen kennen sicherlich das Filter-Element von CSS. Mit diesem verändern Sie das Aussehen eines `img`-Objekts, können dieses beispielsweise alternativ einfärben oder einen Unschärfe-Effekt darauf anwenden. Alles in allem sind die Möglichkeiten jedoch sehr einschränkt. Genau an dieser Stelle setzt das Filter-Element von SVG an. Sie finden in SVG deutlich mehr Filter-Effekte als in CSS. Darüber hinaus bie-

ten diese auch mehr Optionen für die Manipulation eines Bildes an.

Mit dem `<filter>`-Element können Sie diese Effekte Ihren Bildern zuweisen. Zwischen den Tags legen Sie die entsprechenden Primitiven fest, welche den gewünschten Effekt umsetzt. Jede Filter-Primitive führt eine graphische Operation auf ein oder mehrere Objekte durch und liefert ein graphisches Ergebnis. Anhand des Namens können Sie direkt darauf schließen, welche graphische Operation ausgeführt wird.

Unschärfe-Effekt gemäß Gauss

Die im ersten Beispiel verwendete Primitive `feGaussianBlur` nutzt etwa den Unschärfe-Effekt gemäß Gauss. Mehr zur hinterliegenden Berechnungsformel finden Sie in dem W3C Dokument zu den Filter-Effekten. Alle Primitive haben im Übrigen den gleichen Präfix `fe` (filter effect).

Als erstes wird ein Filter mit der ID `filter1` festgelegt. Dieser besitzt eine Standard-Abweichung von jeweils 3px für den x- und den y-Wert. Dieser Filter wird anschließend auf

ein Bild mit dem Namen *bild1.jpg* angewendet. Hierfür weisen Sie innerhalb des *image*-Tags dem Attribut *filter* den Wert *url(#filter1)* zu:

```
<svg width="400px" height="300px">
  <defs>
    <filter id="filter1">
      <feGaussianBlur in="SourceGraphic"
        stdDeviation="3" />
    </filter>
  </defs>

  <image xlink:href="bild1.jpg"
    width="100%" height="100%" x="0" y="0"
    filter="url(#filter1)">
  </image>
</svg>
```

Als Ergebnis erhalten Sie ein Bild, über welches ein deutlich sichtbarer Unschärfe-Filter gelegt wurde ([Bild 1](#)).

Die Filter-Primitiven

Insgesamt gibt es 17 Filter Primitive innerhalb der aktuellen SVG-Filterdefinitionen. Diese liefern Ihnen unter anderem graphische Effekte zu Beleuchtung, Farbe aber auch zur Rausch- und Texturerzeugung.

Vom Prinzip her arbeiten alle Filter nach dem gleichen Schema: die Quell-Grafik wird als Eingabe verwendet und als Ergebnis erhalten Sie eine veränderte Version. Sie können auch mehrere Filter-Effekte nacheinander auf ein Bild anwenden. In diesem Fall ist das Ergebnis des ersten Filters die Eingabe für den zweiten. Durch die Kombination mehrerer Filter-Effekte können Sie eine nahezu unendliche Anzahl an Bildern generieren.

Wie Sie im ersten Beispiel bereits gesehen haben, wird die Eingabe durch den Parameter *in* festgelegt. Sie können für diesen einen oder zwei Werte mitliefern. Als Ergebnis – repräsentiert durch den Parameter *result* – erhalten Sie jedoch immer nur ein Ergebnis. Falls der Filter eine zweite Eingabe erfordert, wird diese über den Parameter *in2* festgelegt.

Wenn Sie mehrere Filter kombinieren und legen im Verlauf keinen Wert für den *in*-Parameter fest, nimmt SVG automatisch das Ergebnis der vorherigen Operation. Anders herum gilt auch: Legen Sie keinen *result*-Wert für das Ergebnis fest, wird dieses automatisch als *in*-Parameter für die nächste Primitive verwendet.

Verschiedene Eingabetypen

Im ersten Beispiel haben Sie bereits einen anderen Typ von Eingabe erlebt: *SourceGraphic*. Dabei erwartet die Funktion ein Element, auf das der Filter angewendet werden soll, etwa ein Bild oder ein Text. Alternativ dazu gibt es noch *SourceAlpha*. Dieser Eingabe-Typ agiert nach dem gleichen Prinzip, allerdings wird von dem Bild lediglich der Alpha-Kanal verwendet. Falls Sie, wie in unserem Beispiel ein JPEG-Bild übergeben haben, erhalten Sie als Ergebnis einen schwarzen Kasten mit den Kantenlängen des übergebenen Bildes. Damit sieht die Definition des Filters wie folgt aus ([Bild 2](#)):

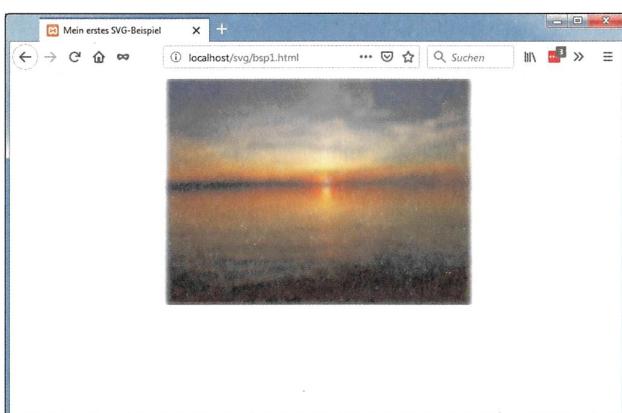
```
<defs>
  <filter id="filter1">
    <feGaussianBlur in="SourceAlpha"/>
  </filter>
</defs>
```

Im nächsten Beispiel zeigen wir Ihnen, wie Sie verschiedene Filter kombinieren können. Ziel ist es, einen individuell gestalteten Schatten zu erzeugen, der nur leicht grau eingefärbt sein soll. Dazu unternehmen wir die folgenden Schritte:

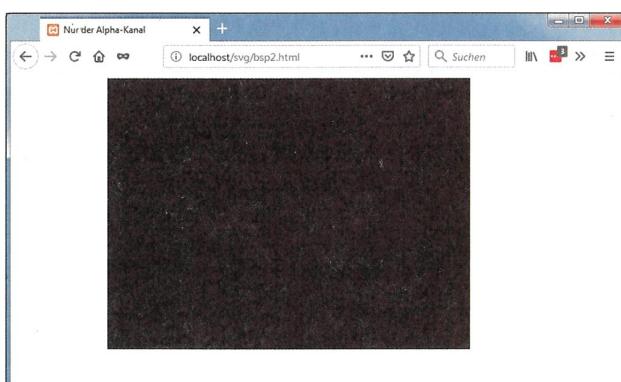
- Für den Schatten verwenden wir den Alpha-Kanal des Bilds und verschieben das Ergebnis um 20 Pixel nach rechts und nach unten.
- Das schwarze Rechteck wird unscharf gezeichnet.
- Der Schatten wird aufgehellt.
- Das Bild und der Schatten werden kombiniert.

In konkreten Code umgesetzt, sieht das wie folgt aus:

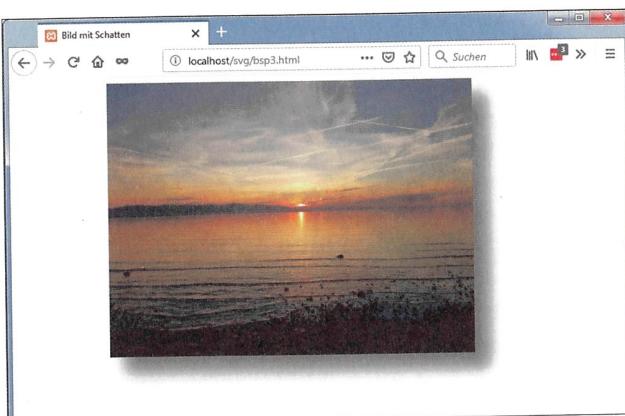
```
<filter id="filter2">
  <feOffset in="SourceAlpha" dx="20" dy="20">
```



Unschärfe: Mit dem Filter *feGaussianBlur* fügen Sie einen Unschärfe-Effekt in Ihr Bild ein ([Bild 1](#))



Nur ein Kanal: Mittels *SourceAlpha* können Sie sich nur den Alpha-Wert des Bildes anzeigen lassen ([Bild 2](#))



Der eigene Schatten: Mit einer Kombination verschiedener Filter generieren Sie einen individuellen Schatteneffekt ([Bild 3](#))

```
</feOffset>
<feGaussianBlur stdDeviation="10" result="unscharf">
</feGaussianBlur>
<feComponentTransfer in="unscharf" result="schatten">
<feFuncA type="table" tableValues="0 0.5"></feFuncA>
</feComponentTransfer>

<feMerge>
  <feMergeNode in="schatten"></feMergeNode>
  <feMergeNode in="SourceGraphic"></feMergeNode>
</feMerge>
</filter>
```

Für die Umsetzung der einzelnen Schritte legen Sie als erstes wieder einen Filter fest – im Beispiel erhält dieser den Namen *filter2*. Den Schatten generieren Sie mit Hilfe der Primitive *feOffset*, die Verschiebung um 20 Pixel realisieren Sie entsprechend mittels der Parameter *dx* und *dy*. Für die Darstellung der Unschärfe nutzen Sie schließlich die bereits bekannte Primitive *fdGaussianBlur* – das Ergebnis erhält den Namen *unscharf*.

Zur Anpassung der Farbintensität des Schattens benötigen Sie zwei Schritte: *feComponentTransfer* bewirkt die Änderung der Helligkeit, des Kontrastes und der Farbbebalance durch eine Änderung der Farbwerte jedes einzelnen Pixels. Da wir für den Schatten lediglich den Alphakanal verwenden haben, benötigen Sie als Kindelement die Primitive *feFuncA*.

Zum Abschluss fügen Sie noch das Bild und den Schatten zusammen. Hierfür nutzen Sie die Primitive *feMerge* mit den Kindelementen *feMergeNode*. Hier geben Sie jeweils das ursprüngliche Bild sowie den generierten Schatten an ([Bild 3](#)).



Umgebender Rahmen: Die Bounding Box stellt den Filterbereich des Objekts dar ([Bild 5](#))

Damit haben Sie mehrere Filter miteinander kombiniert und einen Schatten nach Ihren Vorstellungen generiert.

Filterbereich festlegen

Jeder Filter hat einen Anwendungsbereich für den er festgelegt wird. Dies kann beispielsweise ein Bereich innerhalb eines SVG sein oder aber die Ränder eines Bildes. Die Grenzen der Elemente innerhalb von SVG sind durch die Ränder des Begrenzungsrahmens definiert. Jedes Element wird direkt von einer sogenannten Bounding Box oder auch BBox eingerahmt. Dabei handelt es sich um ein Rechteck, welches direkt an den Außengrenzen des Objekts ansetzt. Im Falle eines Textes orientiert es sich immer am äußersten Punkt. Gerade bei ausladenden Buchstaben – wie im Beispiel des Schriftzugs *Bounding Box!!* – kann so an den Rändern ein größerer leerer Bereich entstehen ([Bild 4](#)).

Wenden Sie also einen Filtereffekt auf ein Objekt wie diesen Text an, wird nur der Bereich innerhalb der Bounding Box verändert. Eine größere Anzahl von Filtern sieht bei der Berechnung jedoch auch die Pixel vor, die knapp außerhalb dieses Bereichs liegen. Aufgrund der Restriktionen der BBox werden diese jedoch nicht mehr berücksichtigt. Dadurch wirkt der Effekt, als wenn er an den Grenzen hart abgeschnitten wurde.

Wenden Sie beispielsweise einen Blur-Effekt auf ein Bild an und legen einen größeren Bereich fest, auf welchen die Unschärfe angewendet werden soll, wird dieser durch die Ränder begrenzt ([Bild 5](#)).

Wenn dies nicht in Ihrem Sinne ist, bleibt Ihnen nur eine Alternative: die Erweiterung der Filter-Region. Hierfür stehen Ihnen verschiedene Attribute zur Verfügung: die x- und die y-Koordinate sowie die Höhe und die Breite.

<filter> besitzt den Wert *filterUnits*, mit dem Sie das Koordinatensystem für die Werte festlegen. Hierfür gibt es zwei Alternativen: *objectBoundingBox* und *userSpaceOnUse*. Der erste Wert ist der Standardwert – in diesem Fall beginnt das Koordinatensystem in der linken oberen Ecke



Abgeschnitten: Abhängig von der Festlegung des Filter-Bereichs wirkt das resultierende Bild an verschiedenen Stellen abgeschnitten ([Bild 4](#))

des Elements und erstreckt sich über die komplette Bounding Box:

```
<filter id="filter" x="5%" y="5%" width="100%" height="100%>
```

In diesem Beispiel ist der Filterbereich die komplette Breite und Höhe des SVG-Bereichs angewendet, allerdings wird der Filter auf der x- und der y-Achse jeweils um fünf Prozent verschoben, bezogen auf die Bounding Box.

Im Fall von `userSpaceOnUse` wird der Wert auf dem existierenden Koordinatensystem übernommen, in welches das Muster eingebunden ist:

```
<filter id="filter" filterUnits="userSpaceOnUse" x="5px" y="5px" width="500px" height="350px">
```

In diesem Fall wird der Bereich auf der x- und y-Achse um jeweils 5 Pixel verschoben und ist absolut gesehen 500 Pixel breit und 350 Pixel hoch.

Filterregion anzeigen lassen

Wenn Ihnen dies nicht greifbar genug erscheint, sollten Sie sich die aktuell gültige Filterregion einfach anzeigen lassen. Dies gelingt über einen kleinen Umweg recht einfach. Setzen Sie hierfür am besten die Primitive `feFlood` ein, mit der Sie einen Bereich einfärben können. Im folgenden Beispiel wird ein Bereich mit einer Größe von 800 auf 400 Pixel festgelegt. Zur Veranschaulichung des Bereichs integrieren wir dorthin den Text *Die Flut!!*. Dieser soll, damit es optisch besser aussieht, 100 Pixel auf der x-Achse und 200 Pixel auf der y-Achse verschoben werden. Damit sieht der Programmcode wie folgt aus:

```
<svg width="800px" height="400px" viewBox="0 0 800 400">
  <text dx="100" dy="200" font-size="144" font-weight="bold" filter="url(#flut)">Die Flut!!
  </text>
</svg>
```

Der Text befindet sich mitten auf der Website, es ist jedoch noch nicht klar, an welcher Stelle die Filterregion startet und wo sie aufhört. Dies bekommen Sie erst heraus, wenn Sie die Filterregion farblich hervorheben. Dazu verwenden Sie, wie bereits angekündigt, die Primitive `feFlood`. Diese wird in das `<filter>` Element integriert:

```
<filter id="flut" x="0" y="0" width="100%" height="100%">
  <feFlood flood-color="#AB9876" flood-opacity=".5">
  </feFlood>
</filter>
```

Der Bezugspunkt ist wieder die linke obere Ecke der BBox und erstreckt sich über die komplette Höhe und Breite. Mit der Primitive `feFlood` wird der Bereich eingefärbt – im Beispiel ein Grauton – und die Farbdurchlässigkeit auf 50 Pro-

zent gesetzt. Damit sehen Sie genau, wo die Bounding Box beginnt und wo sie endet. Natürlich lassen sich die beiden Objekte – Text und farblich hervorgehobener Bereich – auch gleichzeitig darstellen.

Hierfür verwenden Sie die Primitive `feMerge`, die bereits in einem der vorherigen Beispiele zum Einsatz kam. Erweitern Sie Ihren Filter einfach um die folgenden Zeilen:

```
<feMerge>
  <feMergeNode in="flut" />
  <feMergeNode in="SourceGraphic" />
</feMerge>
```

Innerhalb der Primitive gibt es zwei Kind-Elemente welche die Schichten darstellen, die miteinander verknüpft werden sollen: den Text und die Hintergrundfarbe. Als Ergebnis erhalten Sie ein klares Bild über die Größe des Filterbereichs.

Wie Sie einen individuellen Schatten für Ihr Bild gestalten, haben Sie bereits in einem der vorherigen Beispiele gesehen. Ein alternativer Weg zur Darstellung der Farbdurchlässigkeit ist der Einsatz der Primitive `feColorMatrix`. Die Matrix bietet Ihnen einen einfachen Weg, die Farbwerte pro Kanal auf Basis des RGBA (Rot-Grün-Blau-Alpha) Modells anzupassen. Die folgende Definition zeigt die Struktur der Primitive:

```
<filter id="filterx">
  <feColorMatrix type="matrix" values="R 0 0 0 0
                                         0 G 0 0 0
                                         0 0 B 0 0
                                         0 0 0 A 0 .."/>
</filter>
</feColorMatrix>
```

Wenn die Werte für R, G, B und A auf 1 sind, dann wird das originale Foto angezeigt. In diesem Fall sieht die Matrix wie folgt aus:

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

Daraus ergibt sich die folgende Berechnung:

$$\begin{aligned} R &= 1*R + 0*G + 0*B + 0*A + 0 \\ G &= 0*R + 1*G + 0*B + 0*A + 0 \\ B &= 0*R + 0*G + 1*B + 0*A + 0 \\ A &= 0*R + 0*G + 0*B + 1*A + 0 \end{aligned}$$

Wenn Sie also ein Bild grün einfärben möchten, entfernen Sie einfach den Kanal für Rot und Blau. Wenden Sie anschließend den Filter auf Ihr Bild an:

```
<defs>
  <filter id="gruen">
```



Eingefärbt: Mithilfe einer Farbmatrix färben Sie die Bilder individuell nach Ihrem Vorstellungen auf Basis des RGB Modells ein ([Bild 6](#))

```

<feColorMatrix
  type="matrix"
  values="0  00  0  0
         0  10  0  0
         0  00  0  0
         0  00  1  0" />
</filter>
</defs>

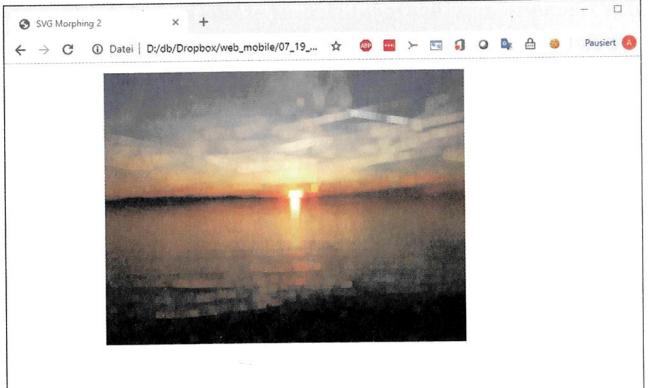
<image xlink:href="bild1.jpg"
width="100%" height="100%" x="0" y="0"
filter="url(#gruen)">
</image>

```

Als Ergebnis erhalten Sie wie gewünscht ein grün eingefärbtes Bild ([Bild 6](#)).

Filter für Bilder und Texte

Ein interessanter Filter, den Sie sowohl für Bilder als auch für Texte einsetzen können, ist *feMorphology*. Übersetzt bedeutet Morphing Verwandlung – der Filter wandelt also das Aussehen eines Objekts um. *feMorphology* besitzt zwei verschiedene Operatoren: *dilate* und *erode*. In der ersten Variante werden die Linienränder einer Grafik verdickt, in der zweiten entsprechend verdünnt. Falls die verwendete Eingangsgrafik keine Struktur besitzt, bleibt sie unverändert. Die Grundstruktur des Filters sieht wie folgt aus:



Verschiedene Parameter: Beim Morphing stehen Ihnen zwei unterschiedliche Optionen zur Verfügung die bestimmen, wie die Übergänge und Farben ineinander übergehen ([Bild 8](#))

dene Operatoren: *dilate* und *erode*. In der ersten Variante werden die Linienränder einer Grafik verdickt, in der zweiten entsprechend verdünnt. Falls die verwendete Eingangsgrafik keine Struktur besitzt, bleibt sie unverändert. Die Grundstruktur des Filters sieht wie folgt aus:

```

<feMorphology
  in=".." result=".."
  operator="dilate || erode" radius="">
</feMorphology>

```

Als weiterer Parameter steht Ihnen noch *radius* zur Verfügung. Damit geben Sie das Umfeld der Operation in Kreisform an. Zulässige Eingaben sind entweder eine oder zwei nicht-negative Zahlen. Bei nur einem Wert gilt dieser für die x- und die y-Richtung. Werden zwei Werte angegeben, ist der erste für die x- und der zweite für die y-Richtung.

Linienränder verwenden

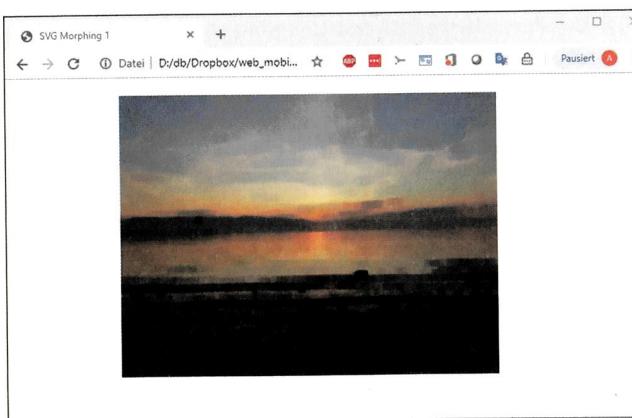
Im folgenden Beispiel wird unser Sonnenuntergang verwendet und die Linienränder entsprechend verdünnt ([Bild 7](#)):

```

<filter id="filter_erode">
  <feMorphology operator="erode" radius="4">
  </feMorphology>
</filter>

<image xlink:href="bild1.jpg"
width="90%" height="90%" x="10" y="10"
filter="url(#filter_erode)">
</image>

```



Morphing: Mittels Morphing können Sie ein Bild verändern und Übergänge verschwimmen lassen ([Bild 7](#))

Bei diesem Radius werden die Pixel im Bild um 4 Pixel geschrumpft. Dadurch erscheint das Bild auch ein wenig kleiner als das Original. Führen Sie die gleiche Primitive aus und nutzen als Alternative den Operator *dilate*, erhalten Sie ein komplett anderes Ergebnis.

Beide Bilder haben sehen nach der Bearbeitung mit dem Filter ein wenig surreal aus. Auf den zweiten Blick bemerken



Texte: Die Filter-Effekte von SVG lassen sich auch auf Texte anwenden ([Bild 9](#))

Sie sicherlich auch die veränderten Farbspektren der beiden Bilder. Das erste Bild, bei dem die Linienränder verdickt wurden, wirkt eher dunkel. Das zweite Bild hat dagegen eine deutlich freundlichere Wirkung. Dafür gibt es aus technischer Sicht auch eine einfache Erklärung: bei *erode* orientiert sich der Bildpunkt an dem dunkelsten oder transparentesten Nachbarn, bei *dilate* entsprechend nach dem hellsten oder am wenigsten transparentesten. Dies führt zu einer dunkleren beziehungsweise helleren Erscheinung ([Bild 8](#)).

Arbeiten mit Texten

Nachdem Sie gesehen haben, welche Effekte Sie mit *feMorphology* bei Bildern erzielen können, wenden wir den Filter in den nächsten Beispielen auf Texte an. Dadurch lassen interessante Effekte wie Schatten oder mehrfarbige Schriftzüge erzeugen. Für unser nächstes Beispiel verwenden wir einen SVG-Text, den wir mit dem Attribute *stroke* verschönern. Damit erhält die schwarze Schrift einen farbigen Rand – in unserem Beispiel mit einer Breite von drei Pixel:

```
<text font-size="80px" dx="100" dy="200" font-weight="700" stroke="lightgreen" stroke-width="3px">
Mein Text-Beispiel</text>
```

Wenn Sie sich den verschönernten Text im direkten Vergleich zum Original anschauen, sehen Sie, dass die Verschönerung des Textes vor allem nach innen wirkt. Gerade bei schmalen Buchstaben, wie dem M oder dem kleinen E wird dies besonders deutlich.



Kombinatorik: Im Zusammenspiel mehrerer Filter erzielen Sie das beste Ergebnis für die individuelle Darstellung Ihrer Texte ([Bild 10](#))

Wenn Sie mehr individuelle Gestaltungsmöglichkeit benötigen, nutzen Sie am besten wieder die SVG Filter und im Speziellen den *feMorphology*. Dazu definieren Sie einen Filter mit dem Operator *dilate* und einem Radius von 4 Pixel. Dies ergibt im Vergleich zum Originaltext eine deutlich breitere Schrift ([Bild 9](#)):

```
<filter id="huebsch">
<feMorphology in="SourceAlpha" result="DILATED"
operator="dilate" radius="4"></feMorphology>
</filter>

<text font-size="80px" dx="100" dy="200"
font-weight="700">Mein Text-Beispiel</text>
<text font-size="80px" dx="100" dy="300"
font-weight="700" filter="url(#huebsch)">
Mein Text-Beispiel</text>
```

Möchten Sie den *stroke*-Effekt nachbauen, müssen Sie die beiden Texte übereinanderlegen und einen der beiden einfarben. Dazu sind innerhalb von SVG allerdings ein paar Schritte notwendig:

```
<svg width="900" height="200" viewBox="100 0 900 200">
<filter id="filter_effekt">
<feMorphology in="SourceAlpha" result="basis"
operator="dilate" radius="4"></feMorphology>
<feFlood flood-color="lightblue" flood-opacity="1"
result="effekt"></feFlood>

<feComposite in="effekt" in2="basis" operator="in"
result="grundfarbe"></feComposite>

<feMerge>
<feMergeNode in="grundfarbe" />
<feMergeNode in="SourceGraphic" />
</feMerge>
</filter>

<text font-size="85px" dx="125" dy="130"
font-weight="700" filter="url(#filter_effekt)">
Mein Beispiel-Text</text>
</svg>
```

Zuerst legen Sie mittels *feMorphology* wieder den verbreiterten Schriftzug an. Dieser soll in unserem Beispiel die Basis bilden. Für den Farbeffekt nutzen wir *feFlood* und weisen diesem Filter eine hellblaue Farbe zu, die ein wenig lichtdurchlässig ist. Mittels *feComposite* führen Sie die beiden Schichten zusammen und erhalten eine neue Schicht hellblau.

Zu diesem Zeitpunkt ist Ihr Schriftzug deutlich verbreitert und erscheint in einem hellen Blau. Damit Sie den zuvor gesehenen Effekt erzielen, gilt es den originalen und den angepassten hellblauen Schriftzug zu kombinieren. Verwenden Sie dazu wie gewohnt *feMerge*. Zum Schluss weisen ►

Sie den neuen Filter Ihrem Text zu und Sie erhalten einen vergleichbaren Effekt, bei dem jedoch deutlich mehr von der schwarzen Farbe erhalten geblieben ist ([Bild 10](#)).

Ausgeschnittene Buchstaben

Im abschließenden Beispiel zeigen wir Ihnen, wie Sie Ihre Buchstaben mit dem Hintergrund verschmelzen lassen. Dazu setzen wir wieder den Alpha-Kanal ein um die Grundrisse der Buchstaben zu erhalten und arbeiten wie zuvor mit mehreren Schichten. Allerdings legen wir nicht mehr wie zuvor den Text oben auf das mit *dilate* behandelte Objekt sondern schneiden es quasi aus:

```
<svg width="900" height="450" viewBox="0 0 900 450">
  <filter id="meinFilter">
    <feMorphology operator="dilate" radius="8"
      in="SourceGraphic" result="THICKNESS" />
    <feComposite operator="out" in="THICKNESS"
      in2="SourceGraphic" /></feComposite>
  </filter>

  <text dx="100" dy="300" filter="url(#meinFilter)"
    letter-spacing="10px">Mein Beispiel-Text</text>
</svg>
```

Mit dem SVG Code erzeugen Sie einen weißen Text auf schwarzem Grund. Den Rest werden wir mit Hilfe von CSS erledigen. Dazu weisen Sie als erstes Ihrem Text das gewünschte Aussehen zu: Schriftart, Schriftgröße und Füllfarbe. Damit haben Sie einen Text mit weißer Füllfarbe und schwarzer Umrandung. In unserem Beispiel soll die Füllfarbe des Textes der Hintergrundfarbe entsprechen. Dazu legen Sie als nächstes die Hintergrundfarbe fest und weisen diese der kompletten SVG-Fläche hinzu ([Bild 11](#)):

```
svg {
  background-color: gold;
}
```

Damit haben Sie den ersten Zwischenschritt erreicht. Natürlich lässt sich dieser Effekt auch auf einen HTML-Text, wie etwa die Überschrift *h2* anwenden:

```
<h2>HTML Beispiel-Text</h2>
```

Dazu definieren Sie einen CSS-Style und weisen diesem den SVG-Filter zu. Dies muss aufgrund der nicht vollständig verfügbaren Kompatibilität für die Webkit-Engine explizit gemacht werden. In unserem Beispiel soll der Text einen blauen Rand erhalten. Die letzten beiden Eigenschaften sind nur optische Anpassungen:

```
h2 {
  -webkit-filter: url(#meinFilter);
  filter: url(#meinFilter);
```



Ausgeschnitten: Die Hintergrundfarbe des Filterbereichs lässt sich einfach als Einfärbung für Ihre Texte verwenden ([Bild 11](#))

```
color: blue;
font-size: 7em;
margin: 2vw;
}
```

Damit haben Sie eine weiße Überschrift mit blauem Rand, die sich wieder deutlich flexibler anpassen lässt als die reine CSS-Formatierung ([Bild 12](#)).

Soll dem Text auf der Website besondere Aufmerksamkeit zukommen, wird die reine Formatierung nicht ausreichen. Die folgenden CSS Styles erlauben Ihnen eine farbliche Anpassung des Hintergrunds. In unserem Beispiel wird der Hintergrund schrittweise vom Wert *gold* nach *blau* transformiert. Hierzu sind innerhalb des Stylesheets ein paar Anpassungen notwendig. Eine Animation besteht immer aus zwei Teilen: den Rahmendaten der Animation sowie die eigentliche Animation. Diese wird über *@keyframes* definiert. In unserem Fall soll sich die Farbe verändern. Der Wechsel von Gold nach Blau findet statt, wenn 50 Prozent der Animationsdauer vorbei sind:

```
@keyframes farbaenderung {
  50% {
    background-color: lightblue;
  }
}
```

Innerhalb der Festlegung von *svg* legen Sie fest, wie die Rahmenparameter der Animation aussehen. Die Animation soll



Alternative Anwendung: Filter-Effekte funktionieren auch problemlos bei reinen HTML-Elementen wie der Überschrift 2 ([Bild 12](#))

Links zum Thema

- W3C SVG 2.0
<https://www.w3.org/TR/SVG/>
- W3C Filter Effekte
<https://www.w3.org/TR/filter-effects>
- Filter auf MDN
<https://developer.mozilla.org/en-US/docs/Web/SVG/Element/filter>
- SVG Filter bei SelfHTML
<https://wiki.selfhtml.org/wiki/SVG/Filter>

zwei Sekunden dauern und aufgrund des Parameters *linear* in gleichbleibender Geschwindigkeit ausgeführt werden. Damit es eine Dauerschleife wird, setzen wir zusätzlich den Parameter *infinite*:

```
svg {  
background-color: gold;  
animation: farbaenderung 2s linear infinite;  
animation-delay: 3s;  
}
```

Damit beim Aufruf der Animation erst einmal die Hintergrundfarbe zu sehen ist, bauen wir eine Verzögerung von drei Sekunden ein. Wenn Sie diese weglassen, startet der Farbwechsel direkt beim Aufruf der Webseite.

Fazit

Wie Sie in den diversen Beispielen, die wir im Artikel behandelt haben, gesehen haben, sind die SVG-Filter ein mächtiges Werkzeug und deutlich leistungsstärker als die Funktionen, die Ihnen unter CSS zur Verfügung stehen.

Jetzt liegt es an Ihnen zu entscheiden, ob Ihnen die Basisfunktionen ausreichen, die mit CSS kommen oder ob Sie gerne mehr Individualität hätten, um die Grafiken oder Texte nach Ihren Vorstellungen zu gestalten. In diesem Fall kommen Sie an den SVG-Filtern nicht vorbei. Sie bieten Ihnen dank der flexiblen Kombinatorik deutlich mehr Gestaltungsspielraum. Die hier vorgestellten Beispiele sind nur ein Anfang.

Sind Sie auf den Geschmack gekommen, empfehle ich Ihnen die Websites von MDN (Mozilla Developer Network) und SelfHTML als weiterführende Literatur. Auf diesen Websites werden Sie sicherlich noch weitere Anregungsmöglichkeiten finden. ■



Andreas Hitzig

arbeitet seit mehr als zwei Jahrzehnten als freiberuflicher IT Autor. Neben dem Thema Web Development sind seine weiteren Schwerpunkte Android und IT Security.



JVMCon
Die Konferenz für Java-Entwickler

25. – 27. November 2019, Köln

SAVE THE DATE
25. – 27.11.2019,
Köln

Themen im Programm

- **Jakarta EE**
- **Reactive**
- **JVM-Languages**
- **Java Virtual Machine (JVM)**
- **Java Runtime Environment (JRE)**
- **Enterprise Java**
- **Cloud Native Development**
- **MicroProfile**
- **Java Security**
- **Java Concurrency**
- **Mobile with Java**
- **u.v.m**

web & mobile DEVELOPER
Leser erhalten 10 % Rabatt
mit dem Code **JVM19wmd**

jvm-con.de



@JVMConference

Präsentiert von:

web & mobile

DEVELOPER

Veranstalter:

 developer
media

AKTUELLES VON BOOTSTRAP

Version 5 im Visier

Seit Bootstrap 4 sind attraktive neue Features dazugekommen.

Bootstrap 4 erschien im Januar 2018 und brachte viele grundlegende Neuerungen – allen voran die Einführung von Flexbox. Inzwischen sind weitere Releases erschienen, die Bootstrap mit neuen Features ausstatten. Die Beliebtheit von Bootstrap ist dabei ungebrochen, was sich auch daran ablesen lässt, dass es eine Version für die Verwendung mit Vue.js gibt ([Bild 1](#)). Apropos Framework – hier verspricht Bootstrap 5 eine wesentlichen Änderung: Es wird ohne jQuery funktionieren und auf JavaScript pur setzen. Aber nun zu den Details der letzten Entwicklungen, den geplanten Features für Bootstrap 5 und BootstrapVue.

Mehr Komponenten

Gerade durch die vielen Komponenten ist Bootstrap so attraktiv – direkt einsatzbereit sind Navigationsleisten, Carousels, Buttons, Dropdowns, Breadcrumbs, Alert-Boxen, die Card-Komponente und eine ganze Menge mehr. Seit Bootstrap 4 sind weitere Komponenten dazu gekommen – ein Spinner, Toast-Nachrichten und ein Schalter in iOS-Optik. Sehen wir uns an, wie diese funktionieren.

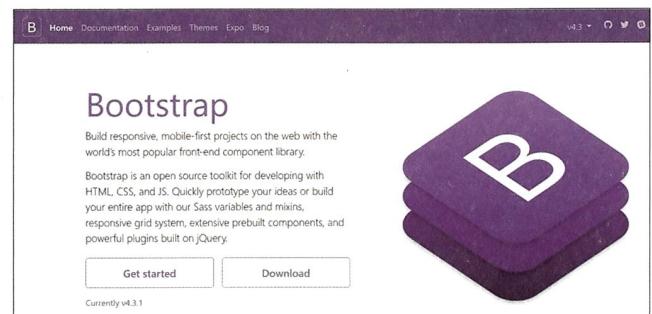
Die Spinner-Komponente wurde in Version 4.2 ergänzt und zeigt einen Ladevorgang an. Es sind zwei Typen von Spinner ([Bild 2](#)) vorgesehen:

- Border-Spinner bestehen aus einer Kreislinie, die nur $\frac{3}{4}$ gezeichnet ist und sich dreht.
- Growing-Spinner setzt auf pulsierende Kreise: Es sind gefüllte Kreise, die wachsen und dabei an Opazität zunehmen, dann verschwinden sie, um von vorne zu beginnen.

Der Code für die Ladeanzeige ist recht kurz. Den Spinner aktivieren Sie durch die Klasse `spinner-border` beziehungsweise `spinner-grow`. Ein Spinner benötigt außerdem die Angabe `role="status"`. Innerhalb des Spinner befindet sich ein `span`-Element mit einem Text für Screenreader. Dieser ist mit der auch sonst bei Bootstrap verwendeten Klasse `sr-only` ausgeteilt und damit in der normalen Ansicht nicht zu sehen:

```
<div class="spinner-border text-primary" role="status">
  <span class="sr-only">wird geladen ...</span>
</div>
```

Das erzeugt einen blauen Border-Spinner. Die Farbe wird durch `text-primary` festgelegt, diese Klasse wird auch sonst beispielsweise bei Buttons zur Definition der Farben benutzt. Weitere mögliche Farben erzeugen Sie durch `text-secondary` (grau), `text-success` (grün), `text-danger` (rot) etc. Wenn Sie die Farben ändern wollen, passen Sie am besten das Farbschema über SASS oder Ähnlich an.



Bootstrap: Die Beliebtheit des Frameworks ist ungebrochen und die Version 5 verspricht zahlreiche Neuerungen ([Bild 1](#))

Wie zu erwarten, wird diese Animation rein mit CSS über eine `@keyframes`-Animation umgesetzt. Für die Spinner-Border-Animation wird nur der Endzustand festgelegt – eine Drehung um 360 Grad.

```
@keyframes spinner-border {
  to { transform: rotate(360deg); }
}
```

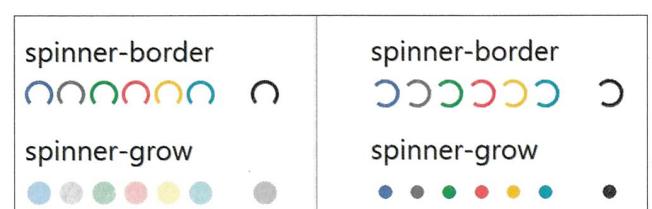
Diese Animation soll linear ablaufen und unendlich oft durchgeführt werden:

```
.spinner-border {
  animation: spinner-border .75s linear infinite;
}
```

Das sind die Angaben im Hintergrund; für die Nutzung des Spinner genügt es, den HTML-Code mit den notwendigen Klassen zu schreiben.

Toast-Nachrichten

Eine Toast-Nachricht ist ein nicht-modales Fenster, um eine kurze Push-Nachricht für den Benutzer einzublenden, die



Zwei Spinner-Varianten: oben Border-Spinner, unten Growing-Spinner ([Bild 2](#))



Toast-Nachricht: So präsentiert sich eine Toast-Nachricht (Bild 3)

von selbst wieder verschwindet. Solche Nachrichten informieren einen beispielsweise, welche neuen Mails gerade gekommen sind.

Der HTML-Code für eine einfache Toast-Nachricht (Bild 3) sieht folgendermaßen aus: Es gibt einen umfassenden Container für die Toast-Nachricht, die aus einem Header und einem Body besteht. Im Header ist neben einer Überschrift ein Button untergebracht.

```
<div class="toast" role="alert" aria-live="assertive"
aria-atomic="true" data-autohide="false">

<div class="toast-header">
  <strong class="mr-auto">Wichtig</strong>
  <button type="button" class="ml-2 mb-1 close"
  data-dismiss="toast" aria-label="Schließen">
    <span aria-hidden="true">&times;</span>
  </button>

</div>
<div class="toast-body">
  Eine Toast-Nachricht
</div>
</div>
```

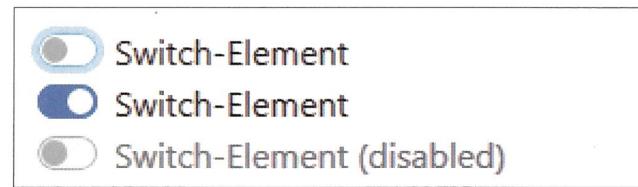
Bei der Toast-Komponente sind ein paar Besonderheiten zu berücksichtigen: Toasts sind aus Performance-Gründen *opt-in*, das heißt, man muss sie explizit initialisieren und sie sind standardmäßig ausgeblendet. Dies kann man über *data-autohide="false"* per HTML oder mit *autohide: false* per JavaScript in den Optionen ändern. Damit die Toast-Nachricht direkt zu sehen ist, wurde oben schon dieses *data*-Attribut verwendet.

Button zum Wegklicken

Wichtig ist der Button, über den die Benutzer die Nachricht wegklicken können. Dieser Button ist mit *data-dismiss="toast"* versehen. Außerdem muss man noch die Toast-Nachricht einblenden lassen:

```
<script>
  $('.toast').toast('show');
</script>
```

Normalerweise wird das als Reaktion auf irgendeinen äußeren Umstand passieren. Der Einfachheit halber wird hier die Toast-Nachricht direkt beim Aufruf der Seite angezeigt. Die Toast-Elemente sind leicht durchscheinend. Zusätzlich wird



Switch: Checkboxen im iOS-Stil (Bild 4)

ein *backdrop*-Filter verwendet, sofern der eingesetzte Browser ihn unterstützt:

```
backdrop-filter: blur(10px);
```

Damit wird der Hintergrund unscharf, ohne dass der Text davon beeinträchtigt wird – ein recht ansprechendes Feature, das derzeit allerdings nur in Safari/iOS-Safari und Edge < 19 implementiert ist. In Chrome funktioniert es ab Version 47 hinter einem Flag, sodass zu erwarten ist, dass man die Backdrop-Filter demnächst auch in Chrome sehen kann.

On-Off à la iOS

Eine weitere kleine neue Komponente ist der Switch im iOS-Stil (Bild 4). Dabei handelt es sich technisch gesehen um individuell gestylte Checkboxen (*<input type="checkbox">*) mit zugehörigen *label*-Elementen. Sie lassen sich wie ein Schalter aktivieren und deaktivieren. Der folgende Code erzeugt einen solchen Switch:

```
<div class="custom-control custom-switch">
  <input type="checkbox" class="custom-control-input"
  id="customSwitch1">
  <label class="custom-control-label"
  for="customSwitch1">Switch-Element</label>
</div>
```

Wenn der Schalter nicht aktiv angezeigt werden soll, ergänzen Sie die Angabe *disabled*.

RFS – Responsive Schriftgrößen

Bootstrap bringt an sich schon einiges mit, was das Erstellen von responsiven Webseiten erleichtert: da wäre zuerst einmal das responsive Raster zu nennen, über das sich unterschiedliche Aufteilungen für verschiedene Viewports definieren lassen. Außerdem gibt es eine Klappnavigation mit Hamburger-Icon für kleine Screens (navBar-Komponente). Und auch Formulare und Tabellen können mit den richtigen Klassen schnell responsiv gestaltet werden. Was bisher aber noch fehlte, war eine Anpassung der Schriftgröße an den Viewport: Idealerweise sollte auf kleinen Geräten die Schrift etwas kleiner sein als bei großem Viewport, damit der hier sehr beschränkte Platz besser genutzt wird. Außerdem halten Benutzer üblicherweise kleinere Geräte näher ans Gesicht als größere: die Entfernung vom Auge zum Bildschirm ist bei einem Smartphone geringer als bei einem Notebook, deswegen darf die Schrift bei einem Smartphone kleiner sein. ►

Man kann für die einzelnen Breakpoints natürlich unterschiedliche Schriftgrößen festlegen; mit dem responsiven Font-Sizing (RFS) von Bootstrap 4.3 geht die Schriftanpassung hingegen automatisch.

Für anpassbare Schriften kommen einem die Viewport-basierten Einheiten `vw` (*viewport width*) und `vh` (*viewport height*) in den Sinn. Wenn man diese für Schriftgrößen verwendet, sind jedoch die Unterschiede zwischen den einzelnen Viewportgrößen zu gewaltig, bei kleinen Viewports fällt die Schriftgröße zu klein aus. Die Lösung besteht darin, `rem` als normale Schriftgrößeneinheit mit Viewport-Einheiten zu kombinieren, beispielsweise so:

```
font-size: calc(1.525rem + 3.3vw);
```

Damit gibt es eine Basisschriftgröße – im Beispiel oben `1.525rem` –, zu der `3.3vw` addiert werden. Die `3.3vw` ergeben je nach Viewportbreite verschiedene Werte und sorgen so für die nötige Flexibilität.

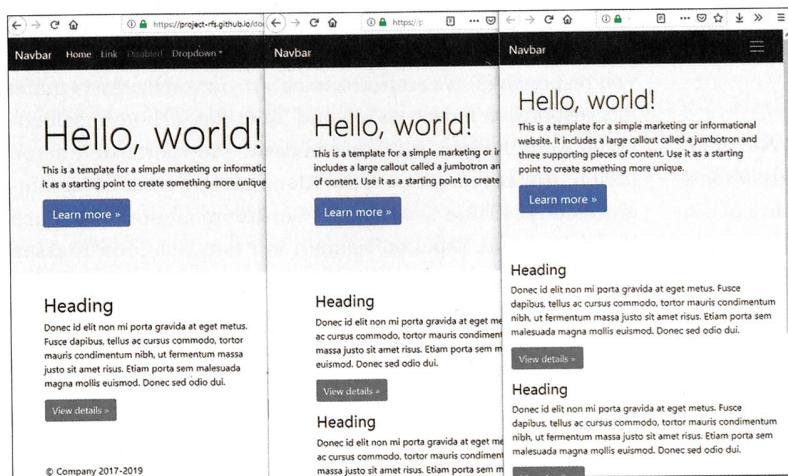
Maximalgröße der Schrift

Bei Bootstrap geben Sie für das responsive Font-Sizing die Maximalgröße der Schrift vor: Die Schrift schrumpft dann bei kleineren Viewports, wird aber nie kleiner als eine definierte Mindestschriftgröße. Legt man beispielsweise für `.title` beim responsiven Font-Sizing eine Größe von `4rem` fest, wird dadurch der folgende CSS-Code erzeugt:

```
.title {
  font-size: 4rem;
}

@media (max-width: 1200px) {
  .title {
    font-size: calc(1.525rem + 3.3vw);
  }
}
```

Ein sehr vernünftiger Ansatz, der die Anpassungen in einem guten Rahmen hält (Bild 5).



Drei verschiedene Viewportgrößen: Mit responsive Font-Sizing passt sich die Schrift an – aber nur in vernünftigen Maßen (Bild 5)

Das responsive Font-Sizing ist in Bootstrap allerdings nicht standardmäßig aktiviert. Um es zu nutzen, brauchen Sie die Bootstrap-Originaldateien, das heißt, nicht die kompilierten CSS-Dateien, sondern die Sources. Diese finden Sie ebenfalls im Download-Bereich von Bootstrap. Wie die Aktivierung von RFS und die Erzeugung der angepassten CSS-Dateien konkret funktioniert, hängt davon ab, ob Sie Stylus, SCSS, Less oder PostCSS nutzen möchten. Hier sehen Sie die SCSS-Variante.

Zuerst benötigen Sie eine eigene SCSS-Datei, in der Sie Bootstrap importieren. Darin müssen Sie die Datei `bootstrap.scss` einbinden, die sich standardmäßig im Unterordner `scss` der heruntergeladenen Quelldateien befindet. Wenn Ihre eigene `scss`-Datei ebenfalls im Ordner `scss` steht, gelingt die Einbindung durch den folgenden Code:

```
@import "bootstrap";
```

Das responsive Font-Sizing aktivieren Sie über eine Variable:

```
$enable-responsive-font-sizes : true;
```

Dann können Sie das responsive Font-Sizing beispielsweise bei einer Klasse nutzen. Sie geben dabei die Größe an, die die Schrift bei großen Viewports haben soll:

```
.title {
  @include font-size(5rem);
}
```

Die scss-Datei sieht also als Ganzes wie folgt aus:

```
@import "bootstrap";
$enable-responsive-font-sizes : true;
.title {
  @include font-size(5rem);
}
```

Dieser Code muss kompiliert werden. Dafür können Sie eines der gängigen SCSS-Tools nutzen, wer lieber eine grafische Oberfläche benutzt, kann beispielsweise Prepros einsetzen. Nach der Umwandlung ist ein Stylesheet mit folgendem Code generiert:

```
.title {
  font-size: 5rem;
}

@media (max-width: 1200px) {
  .title {
    font-size: calc(1.625rem + 4.5vw);
  }
}
```

Nun ist die Schrift bei der Klasse `title` ab einem Viewport von 1.200 Pixeln 5 rem groß, darun-

ter passt sie sich an den Viewport an, wird aber nie kleiner als 1.625rem. Bei der Definition des RFS können Sie statt rem auch Pixelwerte angeben:

```
@include font-size(80px);
```

Diese werden dann bei der Kompilierung in rem umgerechnet. Neben dem gezeigten *font-size*-Mixin sind zwei weitere Varianten möglich:

```
.title2 {  
  @include responsive-font-size(80px);  
}  
.title3 {  
  @include rfs(80);  
}
```

Der Effekt ist immer derselbe. Die responsive Schriftanpassung können Sie durch weitere Angaben genauer steuern. Da wäre erst einmal die Variable *\$rfs-base-font-size*. Hierüber können Sie die Mindestgröße definieren – die responsive Schriftgröße wird nie kleiner als der hier angegebene Wert:

```
$rfs-base-font-size: 1.5rem;
```

Der Standardwert ist 1.25rem. Logischerweise müssen Sie an das *font-size*-Mixin einen Wert übergeben, der größer ist als die Mindestgröße, damit das responsive Font-Sizing eine Auswirkung hat.

Umrechnung in einen rem-Wert

Standardmäßig wird der übergebene Wert in einen rem-Wert umgerechnet, Sie könnten aber auch eine andere Einheit bei *\$rfs-font-size-unit* spezifizieren.

Die responsive Schriftgrößenveränderung funktioniert nur bis zu einem bestimmten Wert – oberhalb des Wertes ist die Schriftgröße fix. Standardmäßig ist die vorgegebene Schriftgröße bei einer Viewportbreite von 1.200 Pixeln erreicht, bei größerem Viewport wächst die Schrift nicht mehr. Diesen Wert verändern Sie mit *\$rfs-breakpoint*. *\$rfs-factor* bestimmt die Stärke der Schriftgrößenveränderung. Je geringer der Faktor, desto weniger Einfluss hat das responsive Font-Sizing bei kleinen Bildschirmen. Der Standardwert ist 10. Wenn Sie daran etwas ändern, sollten Sie das Ergebnis in Ruhe testen. Sehr nützlich ist die Option *\$rfs-two-dimensional*:

```
$rfs-two-dimensional : true;
```

Wenn Sie diese aktivieren, wird nicht nur die Breite (*max-width*), sondern auch die Höhe des Viewports (*max-height*) bei der Media Query für die responsive Schriftgröße herangezogen. Außerdem wird die Schriftgröße nun nicht mehr über rem + vw berechnet, sondern mit rem + vmin. Das Ergebnis von:

```
.title {  
  @include font-size(5rem);  
}
```

```
}
```

... sieht nach der Aktivierung der zweidimensionalen Anpassung wie folgt aus:

```
.title {  
  font-size: 5rem;  
}  
@media (max-width: 1200px), (max-height: 1200px) {  
  .title {  
    font-size: calc(1.625rem + 4.5vmin) ;  
  }  
}
```

Die Einheit vmin bezieht sich entweder auf die Höhe oder auf die Breite des Viewports, je nachdem, was kleiner ist. Das führt dazu, dass sich die Schriftgröße jetzt bei einem Wechsel zwischen Portrait- und Landscape-Modus nicht verändert (denn da ist der vmin-Wert ja derselbe). Wenn Sie hingegen die zweidimensionale Anpassung nicht nutzen, ändert sich die Schriftgröße, wenn die Nutzer ihr Gerät drehen.

Sie können zudem über *\$rfs-class* angeben, dass spezielle Klassen erzeugt werden:

- *\$rfs-class: false*: keine Klassen werden generiert.
- *\$rfs-class: disable*: Dadurch wird eine Klasse *disable-responsive-font-size* angelegt, die Sie bei einem Element angeben können, um RFS für das Element und seine Nachfahren zu deaktivieren.
- *\$rfs-class: enable*: Jetzt können Sie die Klasse *enable-responsive-font-size* bei einem Element ergänzen, um RFS für das Element und Nachfahren zu aktivieren.

Bootstrap beinhaltet viele Utility-Klassen, über die sich Formatierungen von Elementen steuern lassen, ohne dass man selbst CSS-Code schreiben muss. Auch hier sind seit Bootstrap 4.0 weitere dazugekommen: *flex-fill*, *flex-grow-** und *flex-shrink-** erlauben ein Finetunen von Flexitems. Mit *flex-fill* versehene Items teilen sich den Platz unter sich auf, wobei aber der Inhalt berücksichtigt wird: Elemente mit mehr Inhalt werden breiter, mit weniger Inhalt schmäler. Um diese Klassen einzusetzen, muss das umfassende Element durch die Klasse *d-flex* zum Flexcontainer werden:

```
<div class="d-flex ">  
  <div class="flex-fill ">Ganz viel Text, der lang ist  
  </div>  
  <div class="flex-fill">Flexitem</div>  
  <div class="flex-fill">Flexitem</div>  
</div>
```

Dabei wird im Hintergrund folgender Code für *flex-fill* verwendet:

```
flex: 1 1 auto !important;
```

Entscheidend ist die Angabe *auto* für *flex-width*, wodurch die Breite des Elements herangezogen wird. Wenn hingegen ►

wirklich gleich breite Elemente erwünscht sind – unabhängig von der eigentlichen Breite des Inhalts –, kann man auf das Raster-System zurückgreifen. Das Raster gewährleistet, dass die Elemente wirklich nicht breiter werden als vorgegeben. Möglich wird das durch `flex-basis: 0` im CSS-Code, was die ursprüngliche Breite des Inhalts ignoriert.

Wenn ein einzelnes Element einer Reihe von Elementen breiter werden soll, können Sie die Klasse `flex-grow-1` ergänzen; soll hingegen ein einzelnes Element schrumpfen, schreiben Sie `flex-shrink-1`. Diese in 4.1 eingeführten Klassen gibt es ebenfalls in den responsiven Varianten, also beispielsweise auch `flex-sm-fill`, `flex-md-fill`, `flex-lg-fill` und `flex-xl-fill`. Damit gelten diese erst ab der angegebenen Gerätekasse.

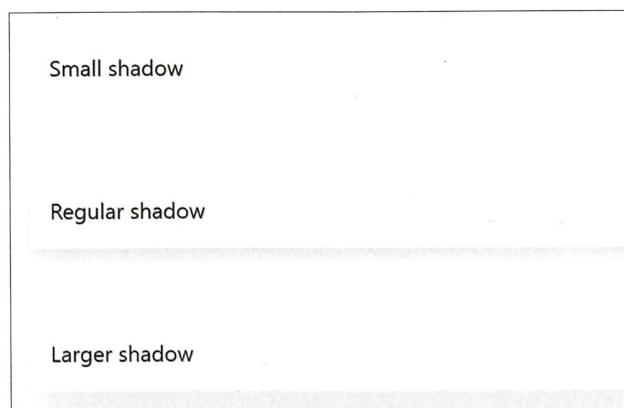
Und es gibt viele weitere neue Klassen, hier eine Auswahl:

- Eine rahmenlose Tabellenvariante können Sie mit `table-borderless` erstellen.
- `rounded-sm` (`border-radius: 0.2rem`) und `rounded-lg` (`border-radius: 0.3rem`) bestimmen die Stärke der abgerundeten Ecken. Der Wert der schon länger vorhandenen Klasse `rounded` liegt mit `0.25rem` dazwischen.
- Zudem gibt es eine Klasse zur Aktivierung einer dichten-gleichen Schrift mit Namen `text-monospace`.
- `text-black-50` und `text-white-50` definieren halb transparenten Text in Schwarz oder Weiß.
- Prinzipiell werden inzwischen standardmäßig keine Schatten mehr in Bootstrap genutzt, außer man setzt die Variable `$enable-shadows` bei der Kompilierung auf `true`. Wenn man trotzdem schnell mal einen Schatten benötigt, kann man inzwischen auf `shadow`, `shadow-sm` oder `shadow-lg` zurückgreifen (Bild 6). Bei den Schatten wie bei der `rounded`-Klasse bezeichnen `sm` und `lg` nicht die Viewportgröße, sondern die Intensität der Formatierung.
- `modal-dialog-scrollable` sorgt für scrollbaren Inhalt in modalen Fenstern.

Außerdem hat Boostrap 4.3.1 einen Patch für eine XSS-Lücke bereitgestellt, es gibt nun einen JavaScript-Sanitizer.

Bootstrap 5

Entscheidende Neuerungen verspricht die kommende Version Bootstrap 5:



Schattenvariationen über `shadow*`-Klassen (Bild 6)

- Von jQuery zu normalem JavaScript: Bisher setzte Bootstrap auf jQuery auf, es ist die wichtigste externe JavaScript-Bibliothek, die zum Einsatz kommt. Ab Version 5 wird jQuery nun durch JavaScript pur ersetzt. Daran arbeitet man schon seit August 2017.

- Von Jekyll zu Hugo: Für die Dokumentation hat Bootstrap auf den statischen Seitengenerator Jekyll gesetzt. Ab Version 5 wird stattdessen Hugo verwendet. Hugo brüstet sich damit, das schnellste Framework der Welt zur Erstellung von Webseiten zu sein und basiert auf Go. Der Grund für diesen Wechsel ist, dass Jekyll bei der lokalen Entwicklung zu langsam ist. Für den raschen Einstieg gibt es übrigens vorkompilierte Binaries von Hugo.

- Weitere Änderungen werden in Erwägung gezogen, sind aber noch nicht entschieden: wie beispielsweise eine Mischung von Gridlayout und Flexbox zur Realisierung des Rasters.

Der Wechsel von Jekyll zu Hugo ist mehr für das Bootstrap-Team selbst relevant – oder für diejenigen, die an der Dokumentation mitarbeiten oder diese intensiv nutzen. Der Umstieg auf normales JavaScript ist hingegen eine Änderung, die alle betrifft.

Der Wechsel von jQuery zu normalem JavaScript ist zeitgemäß: die Gründe, die ursprünglich für jQuery gesprochen haben, sind heute nicht mehr relevant. Zur Erinnerung: jQuery 1 erschien im August 2006, da war gerade der Internet Explorer 7 herausgekommen. Heute sind die Browserunterschiede wesentlich weniger gravierend als damals, DOM-Manipulationen sind in JavaScript einfacher, das Auswählen über CSS-Selektoren gelingt mit `document.querySelector(selectors)` ganz ohne Framework und das Event-Handling ist im Wesentlichen browserübergreifend konsistent.

Webseiten wie <http://youmightnotneedjquery.com/> zeigen, wie sich klassische Aufgaben mit und ohne jQuery lösen lassen – und empfehlen gerade Entwicklern von Bibliotheken, sich zu überlegen, ob sie nicht auf jQuery verzichten können.

Der Vorteil von JavaScript pur liegt in der besseren Performance und in der kleineren Download-Größe: Bei Bootstrap 4.3 bringt `jquery.slim.min.js` alleine eine Größe von 68,28 Kilobyte auf die Waage, dazu kommt noch die Bootstrap-JavaScript-Datei `bootstrap.bundle.min.js` mit 76,79 Kilobyte. In der Demoversion der neuen Bootstrap-Variante ohne jQuery steht diesen zwei Dateien nur eine Datei gegenüber: `bootstrap.bundle.min.js` mit 85,88 Kilobyte. Der Erscheinungsstermin von Bootstrap 5 steht noch nicht fest. Es bleibt zu hoffen, dass es sich nicht so lange hinzieht wie die Veröffentlichung von Bootstrap 4.

BootstrapVue

Die Popularität von Bootstrap zeigt sich gerade auch daran, dass es häufig in anderen Projekten eingesetzt wird. So gibt es beispielsweise BootstrapVue (Bild 7). Mit über 40 verfügbaren Plugins und mehr als 75 benutzerdefinierten UI-Komponenten bietet BootstrapVue eine der umfassendsten Implementierungen von Bootstrap für Vue.js.

The screenshot shows the official Bootstrap + Vue documentation. At the top, there's a navigation bar with links like 'BV', 'Docs', 'Components', 'Directives', 'Reference', 'Misc', and 'Play'. Below the navigation, the title 'Bootstrap + Vue' is displayed. A large, stylized 'V' logo is centered on the page. To the left of the logo, there's a block of text about the project's features and its compatibility with Bootstrap v4.3 and Vue.js v2.6+. At the bottom left, there's a section about 'Vue.js' with a small icon.

BootstrapVue: Eine der umfassendsten Implementierungen von Bootstrap für Vue.js (Bild 7)

Um BootstrapVue zu nutzen, brauchen Sie Vue.js, Bootstrap und PortalVue. Bei letzterem handelt es sich um eine Vue.js-Komponente, durch die man das Template einer Komponente irgendwo im DOM rendern kann. jQuery wird – natürlich – nicht benötigt.

Flexibler Einsatz von BootstrapVue

Sie können BootstrapVue auf die verschiedensten Arten einsetzen, etwa mit VueCLI3 oder mit einem Modul-Bundler. Für erste Tests können Sie auch direkt in der HTML-Datei auf die notwendigen Dateien verweisen.

```
<link rel="stylesheet" href="https://unpkg.com/bootstrap/dist/css/bootstrap.min.css" />
<link rel="stylesheet" href="https://unpkg.com/bootstrap-vue@latest/dist/bootstrap-vue.min.css" />

<script src="https://polyfill.io/v3/polyfill.min.js?features=es2015%2CMutationObserver"
crossorigin="anonymous"></script>
```

```
<script src="https://unpkg.com/vue@latest/dist/vue.min.js"></script>
<script src="https://unpkg.com/bootstrap-vue@latest/dist/bootstrap-vue.min.js">
</script>
```

Dann muss man noch in einem weiteren `script`-Bereich bestimmen, wo Vue.js tätig werden soll – im folgenden Beispiel in einem Element mit `id="app"`:

```
<script>
var app = new Vue({
  el: '#app'
})
</script>
```

Nun sind die Komponenten nutzbar. Der folgende Code zeigt ein Badge (`<b-badge>`) und eine alert-Nachricht (`<b-alert>`):

```
<div id="app">
  <h2>Example heading <b-badge>Neu</b-badge></h2>
  <b-alert show variant="primary">Primary Alert
  </b-alert>
</div>
```

Das waren jetzt zwei Beispiele für sehr einfache Bootstrap-Komponenten, die nur aus CSS-Code bestehen. Aber selbstverständlich sind auch komplexere Komponenten, die JavaScript voraussetzen, mit Bootstrap +Vue.js implementiert. Einzelne Komponenten wie nav sind verbessert worden (enhanced), bietet also mehr Features als die Original-Bootstrap-Komponenten. Praktisch ist der Online-Playground, bei dem Sie BootstrapVue direkt ausprobieren können. Für den echten Einsatz empfiehlt es sich, nicht unnötigen JavaScript-Code für nichtbenutzte Komponenten zu laden.

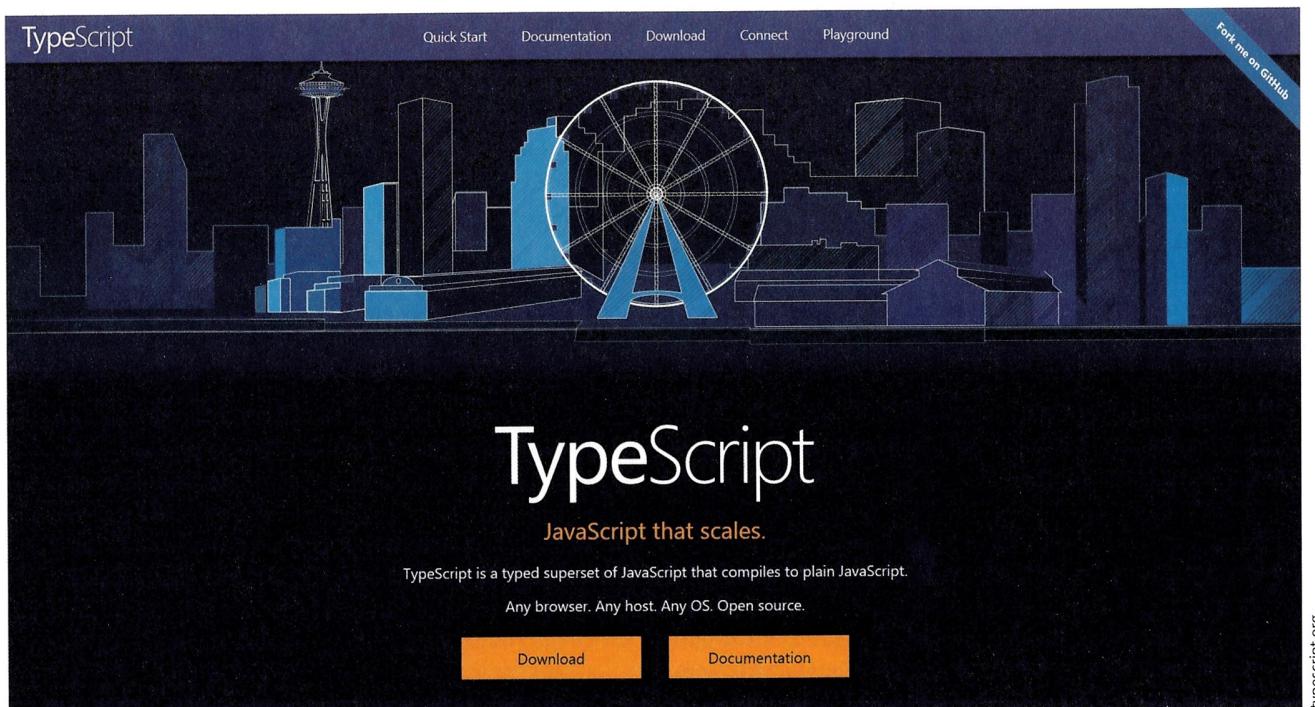
Fazit

Responsives Font-Sizing ist ein sehr attraktives Feature. Auch wer Bootstrap nicht nutzt, kann sich von der Implementierung inspirieren lassen. Bootstrap ohne JavaScript ist auf jeden Fall zeitgemäß und wurde von vielen schon lange gewünscht. Deswegen ist es sehr zu begrüßen, dass Bootstrap 5 dies vorsieht. Dadurch wird BootstrapVue aber nicht überflüssig werden, weil dieses Projekt eine Implementierung von Bootstrap-Komponenten über Vue.js bietet. ■



Florence Maurice

ist Autorin, Trainerin und Programmiererin in München. Sie schreibt Bücher zu PHP und CSS3 und gibt Trainings per Video. Außerdem bloggt sie zu Webthemen unter <https://maurice-web.de/blog/>



UMSTIEG AUF TYPESCRIPT 3.0

Umsteigen lohnt sich

Der Umstieg auf TypeScript bringt für JavaScript-Entwickler einige Vorteile.

Rigidität bei Programmiersprachen ist ein zweischneidiges Schwert. Sie sorgt einerseits dafür, dass der Compiler Fehler erkennt, beschränkt jedoch andererseits den Programmierer beim Implementieren kreativer Konstruktionen. JavaScript ist von Haus aus sehr tolerant – TypeScript legt dagegen Beschränkungen auf. Per se ist die hinter TypeScript stehende Idee nicht neu. Jeremy Ashkenas kam mit Coffeescript auch auf den Gedanken, die JavaScript-Runtime als Ausführungsumgebung für den von einem Transpiler generierten Code einzusetzen ([Bild 1](#)).

Umwandlung in JavaScript

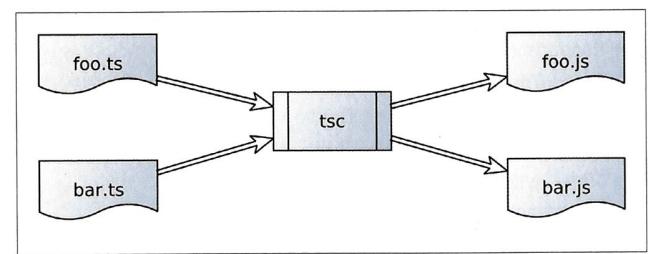
Der Entwickler programmiert dabei in einem mehr oder weniger beliebigen Sprachdialekt, der in gewöhnliches JavaScript umgewandelt wird. Dieser lässt sich dann von Browser und Node-Runtime gleichermaßen ausführen; das einst von Flash und Co. bekannte Herunterladen und Bereitstellen dedizierter Runtimes entfällt ersatzlos.

TypeScript ist im Mobilbereich interessant, weil Telerik es in der hauseigenen NativeScript-Programmierumgebung als Programmiersprache erster Wahl betrachtet. Wir möchten Ihnen hier die Grundlagen eines Workflows auf Basis von TypeScript 3.0 vorstellen.

Da Microsoft die Weiterentwicklung von TypeScript fördert, dürfte es nicht verwundern, dass die Sprache von den

hauseigenen Entwicklungsumgebungen bestens unterstützt wird. Wir werden in den folgenden Schritten mit Visual Studio arbeiten, das unter Ubuntu 18.04 zum Einsatz kommt. Wenn sie die IDE noch nicht auf ihrer Workstation haben, so besuchen Sie die URL <https://code.visualstudio.com/Download> und klicken danach auf den .deb-Knopf.

Die Visual Studio-Webseite stellt Ihnen daraufhin ein rund 50 MByte großes Archiv zur Verfügung, das sie wie gewohnt auf ihrer Workstation installieren. Wer mit Windows oder Mac OS arbeitet, entscheidet sich im in [Bild 2](#) gezeigten Downloadfenster einfach für ein anderes Betriebssystem. Die Visual Studio-Installationsumgebung bringt selbst keinerlei TypeScript-Tooling mit. Die erwähnten Transpilations-Werk-



Die Programmiersprache TypeScript nutzt die JavaScript-Runtime als Ausführungsumgebung ([Bild 1](#))

```
tamhan@TAMHAN18:~$ tsc --version
Version 3.5.1
tamhan@TAMHAN18:~$ 
```

Visual Studio steht für alle wichtigen Betriebssysteme zur Verfügung (Bild 2)

zeuge leben seit längerer Zeit innerhalb der Node.js-Arbeitsumgebung und lassen sich nach folgendem Schema herunterladen:

```
tamhan@TAMHAN18:~$ npm --v
6.9.0
tamhan@TAMHAN18:~$ sudo npm install -g typescript
```

Nach getaner Arbeit sollten Sie durch Eingabe des Befehls TSC überprüfen, ob die Installation problemlos verlief. Bei erfolgreichen Deployments präsentiert sich das Resultat wie in Bild 3. An dieser Stelle sind wir zum Start der Visual Studio-Umgebung bereit. Von Haus aus betrachtet Visual Studio das gerade aktuelle Arbeitsverzeichnis des Terminals als Projektstammordner. Der Autor legt im ersten Schritt einen neuen Arbeitsplatz namens *tsspace* an:

```
tamhan@TAMHAN18:~$ mkdir tsspace
tamhan@TAMHAN18:~$ cd tsspace/
tamhan@TAMHAN18:~/tsspace$ code .
```

Nach dem ersten Start präsentiert Visual Studio ein Begrüßungsfenster, das Sie durch Anklicken des X-Symbols im oberen rechten Tab schließen. Die Genehmigung, Daten an den Microsoft-Server zu übertragen, können Sie nach Belieben gewähren oder verweigern.

TypeScript unterscheidet sich von anderen JavaScript-Arbeitsumgebungen dadurch, dass es dem Entwickler wenige Vorschriften in Bezug auf die Projektstruktur macht. In der Theorie reicht es aus, eine .TS-Datei an den Compiler zu übergeben, der daraus ein JavaScript-File erzeugt.

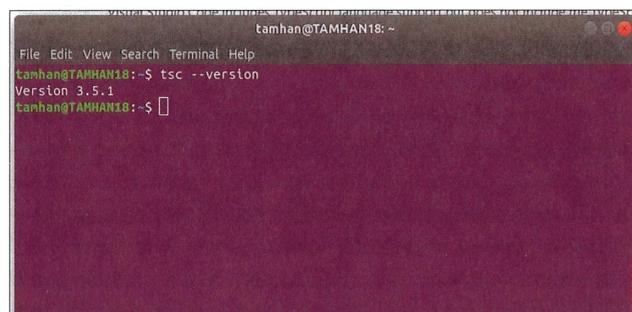
In der Praxis ist es empfehlenswert, die von Microsoft vorgegebenen Standardeinstellungen zu modifizieren. Dies erfolgt über eine JSON-Datei, die prinzipiell auf den Namen *tsconfig.json* hören muss. Klicken Sie im ersten Schritt auf *File* und *New File*, um eine neue Datei mit dem Namen *Untitled-1* anzulegen. Klicken Sie danach auf *File* und *Save As*, um einen Common-Dialog auf den Bildschirm zu holen. Geben Sie dort den Dateinamen *tsconfig.json* ein und klicken Sie danach auf *Save*, um den Korpus der Konfigurationsdatei für unser Projekt anzulegen. Befüllen Sie die noch leere Datei im nächsten Schritt mit folgendem Markup:

```
{
  "compilerOptions": {
    "target": "es5",
```

```
    "module": "commonjs",
    "outDir": "out",
    "sourceMap": true,
    "incremental": true
  }
}
```

Unser Konfigurationsfile ist aus Sicht des Editors eine JSON-Datei wie jede andere. Visual Studio unterstützt Sie bei der Bearbeitung der Compiler-Einstellungen allerdings mit Vorschlägen, die die TypeScript-Spezifikation berücksichtigen.

Das vorliegende File befiehlt die Nutzung eines Unterverzeichnisses, in dem die transpilierten Dateien abgelegt werden. Durch Setzen des Parameters *Source Map* weisen wir *tsc* dazu an, während der Kompilation die für den Debugger not-



Die TypeScript-Runtime ist einsatzbereit (Bild 3)

wendigen Metadaten zu erzeugen. Unterbleibt dies, so können die diversen Fehlersuchwerkzeuge nur den transpilierten JavaScript-Code, nicht aber den für die jeweilige Zeile verantwortlichen TypeScript-Code präsentieren.

Zu guter Letzt setzen wir noch das *Incremental*-Flag auf True. Es handelt sich dabei um eine von TypeScript 3.4 neu eingeführte Einstellung, die das teilweise Rekompilieren von TypeScript-Code erlaubt und so für schnellere Transpilations-Prozesse sorgt.

Im nächsten Schritt müssen wir die erste TS-Datei anlegen. Klicken Sie abermals auf *File* und *New File*, um eine weitere *Untitled-1*-Datei anzulegen. Klicken Sie danach sofort auf *File* und *Save* und vergeben Sie den Namen *Hallo.TS*, um die Datei zu sichern. Es handelt sich dabei übrigens um keine Formalität des Autors: Visual Studio erkennt in Untitled-Dateien wegen des Fehlens des Dateiendung nicht, um ►

welche Art von Quellcode es sich handelt. Aus der Logik folgt, dass die IDE in diesem Betriebsmodus IntelliSense und sonstige Unterstützungen deaktiviert. Es ist also in Ihrem eigenen Interesse, neu angelegte Dateien so schnell wie möglich auf die Festplatte zu speichern.

Im nächsten Schritt platzieren wir den folgenden Code in der Datei:

```
function stringBauer(x: string) {
    return "Die NMG sagt " + x;
}
let hallo: string = "Hallo Welt";
console.log(stringBauer(hallo));
```

Der vorliegende Code ist vom Umfang hier etwas höher als man es von normalen Start-Beispielen erwartet. Im ersten Schritt lege ich eine Funktion an, die einen Parameter vom Typ *String* entgegennimmt.

JavaScript-erfahrenen Entwicklern fällt sofort auf, dass der Wert von X mit einer Typdeklaration ausgestattet ist. Der nach dem Doppelpunkt folgende Typ informiert den Transpiler darüber, dass an dieser Stelle nur ein bestimmter VariablenTyp zugelassen ist.

Zuweisung einer Konstante

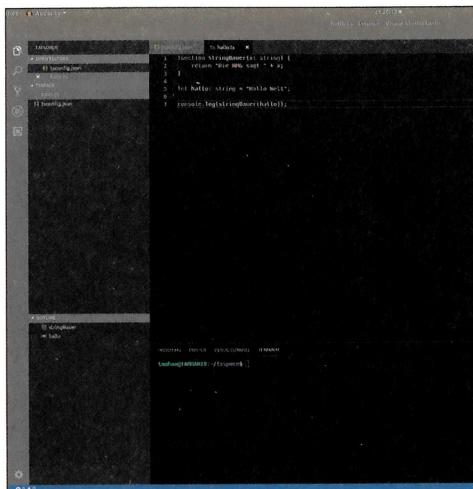
Im nächsten Schritt erzeuge ich eine lokale Variable, die über den Doppelpunkt ebenfalls den Typ *String* zugewiesen bekommt. In der Praxis kann diese Deklaration auch wegfallen, da der Transpiler bei der Zuweisung einer Konstante den vorliegenden Datentyp selbsttätig erkennt. Zu guter Letzt rufe ich dann noch *Console.Log* auf, um die Ausgabe meines kleinen Beispielprogramms auf den Bildschirm zu bringen.

Beachten Sie, dass sich TypeScript an dieser Stelle case sensitive zeigt. *string* (mit kleingeschriebenem s) und *String* mit großgeschriebenem S sind zwei komplett verschiedene Datentypen. Der kleine String ist ein primitiver Datentyp, während der große String eine Wrapper-Klasse darstellt.

Im nächsten Schritt müssen wir das Terminal auf den Bildschirm holen. Dazu klicken wir im Menü auf die Option für *New File Terminal*, woraufhin am unteren Rand des Bildschirms in Bild 4 gezeigte Terminal erscheint.

Wer unter Windows arbeitet, bekommt an dieser Stelle eine PowerShell-Instanz zur Verfügung gestellt. Die ist ein komplett normales Verhalten der IDE, über das sie sich nicht weiter aufregen müssen.

Da unsere Version der Konfigurationsdatei keine Regular Expressions



Das in Visual Studio enthaltene Terminal ist startbereit (Bild 4)

zum Anziehen der Dateien aufweist, wollen wir die zu transpilierenden Dateien anfangs von Hand anmelden:

```
tamhan@TAMHAN18:~/tsspace$ tsc
hallo.ts
tamhan@TAMHAN18:~/tsspace$
```

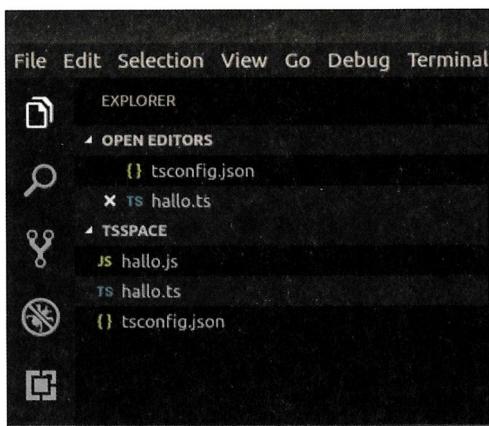
Nach getaner Arbeit präsentiert sich die IDE wie in Bild 5.

Visual Studio zeigt sich beim Dateibaum eigenwillig: Neben dem Ordner *tsspace*, der die im Projekt befindlichen Dateien auflistet, gibt es mit der Rubrik *Open Editors* auch ein virtuelles Verzeichnis. In ihm finden sich all jene Elemente, die im Moment in einem Editor-Tab zur Bearbeitung geöffnet sind. Bevor wir uns den Problemen mit der Ordnerstruktur zuwenden, wollen wir allerdings noch einen Blick in die Datei *Haloo.JS* werfen. Klicken Sie sie doppelt an, um sie in einem Editorfenster zu öffnen:

```
function stringBauer(x) {
    return "Die NMG sagt " + x;
}
var hallo = "Hallo Welt";
console.log(stringBauer(hallo));
```

Nach der Transpilation der TypeScript-Datei ist nichts mehr vom ursprünglichen Code übrig geblieben. Wir haben es hier mit einer reinen JavaScript-Datei zu tun. Daraus ergibt sich eine interessante Feststellung. Die Typhaltigkeitsprüfung erfolgt ausschließlich während der Kompilation - es ist (mit Tricks) möglich, sie zur Laufzeit zu überlisten. Wie dem auch sei, wollen wir uns noch von der Ausführbarkeit des Codes überzeugen. Hierzu reicht es aus, die Node.js-Ausführungs-Umgebung auf die generierte Datei loszulassen. Dies ist insofern kein Problem, als das Terminal-Tab von Visual Studio ja ein vollwertiges Terminal zur Verfügung stellt:

```
tamhan@TAMHAN18:~/tsspace$ node
hallo.js
Die NMG sagt Hallo Welt
```



Die TS-Datei bekam zusätzlich eine JavaScript-Kollegin (Bild 5)

Unsere vorher angelegte JSON-Datei sollte die Konfiguration des Projekts unter anderem insofern beeinflussen, als die Ausgabedateien im Unterordner */out* zu liegen kommen sollten. Leider ist dies aufgrund einer Besonderheit des TypeScript-Compilers nicht der Fall. Da es sich dabei um einen der häufigsten Anfängerfehler handelt, wollen wir ihn Schritt für Schritt abarbeiten. Im ersten Schritt müssen wir allerdings die

schon erzeugte Datei löschen. Unter Linux reicht es dabei aus, den Befehl `rm` in das Visual Studio-Terminal einzugeben:

```
tamhan@TAMHAN18:~/tsspace$ rm hallo.js
tamhan@TAMHAN18:~/tsspace$
```

Im Interesse der Datensicherheit sei noch mal darauf hingewiesen, dass das Terminal in Visual Studio potentiell gefährlich ist. Es gibt nichts, was Sie daran hindert, über ein fehlgeschlagenes Kommando den gesamten Inhalt des Projektordners ins digitale Nirvana zu befördern. Im nächsten Schritt befehlen wir einen weiteren Kompilationslauf, der diesmal aber ohne Parameter auskommen muss:

```
tamhan@TAMHAN18:~/tsspace$ tsc
```

Nach getaner Arbeit präsentiert sich die Projektstruktur wie in [Bild 6](#).

Auswertung der Konfigurationsdatei

Der TSC-Compiler wertet die Inhalte der Konfigurationsdatei `tsconfig.json` nur dann aus, wenn er komplett ohne Parameter aufgerufen wird. In dem Moment, in dem sie einen Dateinamen übergeben, arbeitet er sonst mit den von Microsoft vorgegebenen Einstellungen.

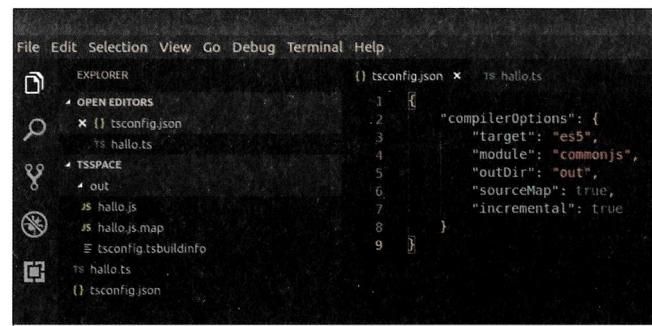
Im gezeigten Ausgabeordner sehen wir neben der JavaScript-Datei noch eine `.map`-Datei, die dem Debugger die Korrelation ermöglicht. Die Datei `tsconfig.tsbuildinfo` ist für den teilweisen Kompilationsprozess verantwortlich. Sie speichert Hashes aller zur Kompilation verwendeten Dateien und erlaubt dem Transpiler so das Finden von Änderungen.

Im nächsten Akt wollen wir zur Datei `Hallo.TS` zurückkehren. Sie enthält die Methode `stringBauer` momentan als frei herumfliegende Funktion. Es wäre im Interesse der Wartbarkeit schöner, wenn wir sie in eine Klasse verpacken könnten. Hierzu ist folgende Anpassung erforderlich:

```
class NMGLager{
    static stringBauer(x: string) {
        return "Die NMG sagt den String " + x;
    }
    static stringBauer(x: number) {
        return "Die NMG sagt die Zahl " + x;
}
```

Die Klassensyntax von TypeScript ist bis zu einem gewissen Grad von C# und Co. inspiriert. Als ersten Versuch erzeugen wir hier eine Klasse namens `NMGLager`, die zwei Methoden mit identischem Namen und unterschiedlichen Parametern enthält. Alte Hasen der objektorientierten Programmierung denken an dieser Stelle mit Sicherheit instinktiv an das Pattern der überladenen Funktion.

Wechseln Sie im nächsten Schritt abermals in das Terminal, speichern Sie die Datei und befehlen Sie danach eine Kompilation. Sie scheitert mit dem in [Bild 7](#) gezeigten Fehler. Ein nettes Komfortfeature ist, dass Visual Studio die seiner Mei-



Der Ordner `out` hat weitere Dateien aufgenommen ([Bild 6](#))

nung nach für das Problem verantwortlichen Teile des Programms auch in der Outline farblich hervorhebt.

Zur Wiederherstellbarkeit der Kompilierbarkeit reicht es anfangs aus, nur eine Version der Funktion anzulegen. Da unser Beispiel bisher mit einem String-Parameter arbeitete, wollen wir die Klasse folgendermaßen vereinfachen:

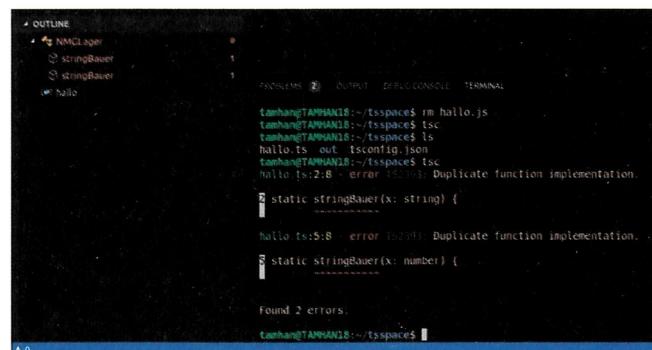
```
class NMGLager{
    static stringBauer(x: string) {
        return "Die NMG sagt den String " + x;
    }
}
```

Im nächsten Schritt müssen wir die soeben korrigierte Klasse noch auf die korrekte Art und Weise aufrufen. Weil wir hier mit einem statischen Element arbeiten, sind Konstruktor und Co noch nicht erforderlich:

```
let hallo: string = "Hallo Welt";
console.log(NMGLager.stringBauer(hallo));
```

An dieser Stelle können Sie abermals zur Kompilation übergehen und den Code danach ausführen. Achten Sie allerdings darauf, dass Sie Node.js nun darüber informieren müssen, dass sich die auszuführende JavaScript-Datei im Unterordner `out` befindet.

Die Implementierung des Overload-Patterns in TypeScript ist ein durchaus interessantes Thema, an dem sich Entwickler und Autoren immer wieder abarbeiten. So sehr das De- ►



Der TypeScript-Transpiler informiert mit farbenfrohen Fehlermeldungen über Probleme ([Bild 7](#))

sign Pattern interessant ist, so gering ist seine Priorität in der Praxis. Wer ordentlich strukturierten Code erzeugen möchte, profitiert von Interfaces wesentlich mehr. Aus der Logik folgt, dass TypeScript auch diese Art von Datentyp unterstützt - es handelt sich ja um ein primär zur Kompilationszeit relevantes Objekt, das zur Laufzeit nur wenig Zusatzaufwand verursacht. Um unseren Interfaces etwas Fleisch zu verpassen, müssen wir in das Typ-System einsteigen. TypeScript leidet dabei unter einer Einschränkung der zugrunde liegenden JavaScript-Runtime. Alle numerischen Typen sind prinzipiell Gleitkommazahlen, da die in den Browsern befindlichen Laufzeit-Umgebungen mit anderen Zahlenarten nichts anzufangen wissen.

Neben Strings, booleschen Werten und anderen aus JavaScript bekannten Typen bietet TypeScript auch einige Komposittypen wie Enums und Arrays an. Sonst sei hier auf die URL <https://www.typescriptlang.org/docs/handbook/basic-types.html> verwiesen, wo die im Allgemeinen selbst erklärenden Typen der TypeScript-Runtime aufgelistet sind.

Spracherweiterungen in Version 3

Leider ist die Dokumentation insofern inkomplett, als Microsoft in Version 3.0 Spracherweiterungen am Typsystem vorgenommen hat. Zum Verständnis ebendieser müssen wir uns vor Augen führen, dass TypeScript-Code nur in den seltensten Fällen leer im Raum steht. Die Routinen müssen so gut wie immer mit gewöhnlichem JavaScript-Code interagieren, der naturgemäß nichts von Typen weiß.

Zur Lösung dieses Problems hat Microsoft Konversionstypen eingeführt: *any* ist von der aus Visual Basic bekannten Variant-Klasse abgeleitet, die mehr oder weniger beliebige Inhalte aufnehmen kann. Als Beispiel dafür dient das folgende Snippet aus der Dokumentation, das einer *any*-Variable verschiedene Datentypen zuweist:

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

any -Variablen unterscheiden sich vom ebenfalls in TypeScript enthaltenen Objekt-Typ dadurch, dass ein *any* -Objekt auch mehr oder weniger beliebige Funktionen enthält und als Ziel für Funktionsaufrufe dienen darf:

```
let notSure: any = 4;
notSure.ifItExists();
// okay, ifItExists might exist at runtime
notSure.toFixed();
// okay, toFixed exists (but the compiler doesn't check)
```

Aus der Logik folgt allerdings, dass das Aufrufen einer inexistenten Memberfunktion einer *any*-Instanz zu einem Laufzeitfehler führt. Verwendet ein Code sehr viele *any* -Objekte, so ist die von TypeScript zur Verfügung gestellte Typsicherheit zumindest teilweise ausgehebelt. Warum man als Entwickler dann den Aufwand treibt, wird ab einer gewissen Stelle fragwürdig.

Mit TypeScript 3.0 führte Microsoft den *unknown*-Typ ein, um das Einfließen von nicht typkontrollierten Werten in TypeScript-Code zu beschränken. Eine *unknown*-Variable nimmt zwar jeden beliebigen Wert auf, lässt sich ohne Cast und Co. aber nur an ein anderes *unknown*-Feld übergeben oder in eine Variable vom Typ *any* speichern.

Aus dieser Situation ergeben sich einige Besonderheiten. Erstens lassen sich *unknown*-Variablen nur mit dem Gleichheits- oder dem Ungleichheitsoperator bearbeiten - die folgenden Operationen würden durch die Bank einen Kompilationsfehler auslösen:

```
function f10(x: unknown) {
    x >= 0; // Error
    x + 1; // Error
    x * 2; // Error
    -x; // Error
    +x; // Error
}
```

Zur Verhinderung der soeben angesprochenen Problematik der Dereferenzierung eines nicht existierenden Memberobjekts greift Microsoft ebenfalls zum Vorschlaghammer. Jeglicher Versuch, ein Member einer *unknown*-Variable zu extrahieren, beendet die Transpilation:

```
function f11(x: unknown) {
    x.foo; // Error
    x[5]; // Error
    x(); // Error
    new x(); // Error
}
```

Der einzige Weg, um *unknown*-Variable auf gewöhnliche Art und Weise nutzbar zu machen, ist ein Typecast. Hierbei empfiehlt es sich, nach folgendem aus der Dokumentation entnommenen Schema vorzugehen:

```
function f20(x: unknown) {
    if (typeof x === "string" || typeof x === "number") {
        x; // string | number
    }
    if (x instanceof Error) {
        x; // Error
    }
    if (isFunction(x)) {
        x; // Function
    }
}
```

Nach diesem kurzen Exkurs in die Welt von Systemen wollen wir uns wieder der Implementierung des in der Einleitung angesprochenen Interfaces zuwenden.

Interfaces können in TypeScript alle öffentlichen Elemente einer Klasse vorschreiben. Es ist sowohl das Festlegen von Membervariablen als auch das Festlegen von Memberfunktionen möglich. Zur Flexibilisierung des weiter oben verwen-

```

1  interface TalkerIf{
2      name: string;
3      zahl?: number;
4      stringBauer(d: string): void;
5  }
6
7
8  class NMGLage implements TalkerIf{
9      name: string;
10     zahl?: number;
11     static stringBauer(x: string): string {
12         return "Die NMG sagt den String " + x;
13     }
14 }
15 let hallo: string = "Hallo Welt";
16 console.log(NMGLage.stringBauer(hallo));
17

```

Hinter dieser Glühbirne verbergen sich diverse Unterstützungs-funktionen (Bild 8)

deten Ausgabecodes möchte ich im ersten Schritt das folgende Interface vorstellen:

```

interface TalkerIf{
    name: string;
    zahl?: number;
    stringBauer(d: string): void;
}

```

Das Interface *TalkerIF* schreibt seinen Implementierern insgesamt drei Member vor. Neben zwei Variablen verlange ich auch das Anlegen einer Funktion, die im Interface vollständig - also auch unter Berücksichtigung der Parameter - definiert werden muss. Das Fragezeichen hinter der Variable *zahl* weist die Runtime darauf hin, dass die Erfüllung dieser Variable beziehungsweise dieser Bedingung nicht unbedingt erforderlich ist. Ein Nutzer des Interfaces könnte also auch einen Talker implementieren, der keine Zahl speichert.

Unsere bisherige und nur mit statischen Attributen ausgestattete Klasse kann die Bedingungen des Interfaces natürliche nicht erfüllen. Visual Studio unterstützt uns allerdings bei der Realisierung der diversen Member. Im ersten Schritt adaptieren wir die Klassendefinition insofern, als wir über das *implements*-Statement Implementierungsabsicht anmelden:

```

class NMGLage implements TalkerIf{
    static stringBauer(x: string) {
        return "Die NMG sagt den String " + x;
    }
}

```

Wer je mit C# gearbeitet hat, sieht an dieser Stelle sofort, dass TypeScript bis zu einem gewissen Grad von Microsofts .net-Vorzeigesprache beeinflusst ist. Legen Sie im nächsten Schritt den Cursor auf das Wort *NMGLage* - es erscheint rot unterstrichen, weil Visual Studio permanent (also während der Arbeit, unabhängig vom Speichern der Datei) eine Syntaxüberprüfung durchführt.

Sobald Sie den Cursor am Wort platziert haben, erscheint - wie in Bild 8 gezeigt - ein Glühlampensymbol über der Cursorzeile. Dieses informiert uns darüber, dass Visual Studio

zum gerade markierten Element Unterstützungen, Refactorings oder sonstige Optionen anbietet. Haarig ist dabei nur, dass diese Optionen normalerweise nur dann erscheinen, wenn sich der Cursor direkt im betreffenden Wort befindet. Legen Sie ihn auf den Operatoren *class* oder *implements* ab, so bietet Visual Studio die Funktion nicht an.

Klicken Sie die Glühbirne an und entscheiden Sie sich im daraufhin erscheinenden Kontextmenü für die Option zur Implementierung des Interfaces. Visual Studio fügt die fehlenden Elemente daraufhin stur in den Kopf des Klassenkorpus ein. Die schon vorhandene statische Funktion müssen sie von Hand entfernen, um folgenden Korpus zu erhalten:

```

class NMGLage implements TalkerIf{
    name: string;
    zahl?: number;
    stringBauer(d: string): void {
        throw new Error("Method not implemented.");
    }
}

```

TypeScript zeigt sich an dieser Stelle insofern eigenwillig, als die Deklaration von Membervariablen keinen Sonderoperator voraussetzt. Es reicht stattdessen, Name und Datentyp durch einen Doppelpunkt getrennt in die Klassendeklaration einzufügen.

Offensichtlich implementiert Visual Studio von Haus aus nicht nur die verpflichtenden, sondern auch die optionalen Member. Neu angelegte Methodenkörper werden analog zu Visual C# mit Code ausgestattet, der bei einem Aufruf eine Abart der *NotImplementedException* wirft.

Im nächsten Schritt möchte ich eine kleine reale Implementierung der Funktion anlegen. Sie kombiniert den vom Entwickler angelieferten String mit dem Inhalt des *name*-Felds und gibt das Resultat in die Konsole aus:

```

class NMGLage implements TalkerIf{
    name: string;
    stringBauer(d: string): void {
        console.log("Habe D " + d + " und den Namen " + name);
    }
}

```

An dieser Stelle unterwelt Visual Studio den Aufruf der Funktion *console.log*. Dies ist insofern logisch, als unsere Klasse nun ja keine statischen Attribute mehr aufweist. Zur Wiederherstellung der Kompilierbarkeit reicht es allerdings aus, nach folgendem Schema eine neue Instanz anzulegen und diese zum Aufruf von *stringBauer* zu verwenden:

```

let hallo: string = "Hallo Welt";
let myinstance: NMGLage = new NMGLage();
myinstance.name="Hallo";
console.log(myinstance.stringBauer(hallo));

```

An sich findet sich hier nichts besonders Interessantes. TypeScript stattet Objekte mit einem Default-Konstruktor ►

aus, der eine Instanz bereitstellt. Da `name` ein öffentliches Element ist, können wir ihm von Hand einen zusätzlichen Wert einschreiben. Danach ist unser Programm auch schon zur Ausführung bereit:

```
tamhan@TAMHAN18:~/tsspace$ node out/hallo.js
/home/tamhan/tsspace/out/hallo.js:5
console.log("Habe D " + d + "und den Namen " + name);
```

Unsere aus optischer Sicht perfekt funktionierende Methode erweist sich in der Praxis als nicht lebensfähig. TypeScript setzt bei Zugriffen auf Membervariablen die Qualifikation `this` voraus. Eine funktionierende Version der Methode würde folgendermaßen aussehen:

```
class NMGLage implements TalkerIf{
  name: string;
  stringBauer(d: string): void {
    console.log("Habe D " + d + "und den Namen " +
    this.name);
  }
}
```

TypeScript verlangt bei Memberzugriffen prinzipiell das Voranstellen des `this`-Operators – ein Entgegenkommen an sonstigen JavaScript-Code, der den globalen Namensraum mitunter mit diversen Objekten vergiftet.

Kompilation und Ausführung erfolgen dann nach folgendem Schema:

```
tamhan@TAMHAN18:~/tsspace$ node out/hallo.js
Habe D Hallo Weltund den Namen Hallo
undefined
```

Interessant ist hier, dass neben der Ausgabe der Funktion `stringBauer` auch die Meldung `undefined` am Bildschirm erscheint. Es handelt sich hierbei um ein Überbleibsel aus dem vorhergehenden Testharrnisch - wir rufen ja `console.log` nochmals auf. Da die Methode `stringBauer` nun aber einen `void`-Wert zurückliefert, erscheint eine leere Ausgabe im Terminal.

Umleitung per Interface

Nachdem wir unsere Klasse bisher noch über eine streng typisierte Variable angesprochen haben, wollen wir die Flexibilität nun erhöhen. Hierzu müssen wir den Typ von `myinstance` auf `TalkerIf` ändern. Quasi nebenbei passen wir auch den `console.log`-Aufruf an, um nun den (in `NMGLage` nicht vorhandenen) Parameter `zahl` in die Debuggerkonsole zu werfen:

```
let myinstance: TalkerIf = new NMGLage();
myinstance.name="Hallo";
console.log(myinstance.zahl);
```

Die JavaScript-Runtime rettet uns an dieser Stelle insofern, als der Zugriff nur den Wert `undefined` zurückgibt. Damit ist

Wieso sind die Listings formatiert

Visual Studio Core hat die unangenehme Eigenschaft, beim Kopieren von Code in der Zwischenablage diverse Formatierungsattribute mitzuliefern. Diese bringen Textverarbeitungen wie Word aus dem Tritt. Möchten Sie ein formatiertes (also mit Whitespaces ausgestattetes) Snippet in lieber Office oder Word übernehmen, so platzieren sie es im ersten Schritt in einem Texteditor. Kopien Sie es von dort abermals in die Zwischenablage, um die Operation abzuschließen.

übrigens auch erklärt, warum Funktionen nicht als optionale Member deklariert werden dürfen. Wäre `zahl` eine Funktion, so gäbe es einen Absturz.

Interfaces erhöhen die Flexibilität des JavaScript-Codes immens. Die Nutzung einer komplett neuen Implementierung von `TalkerIf` setzt – neben der Implementierung – nur das Anpassen des aufgerufenen Konstruktorfunktion voraus:

```
class NMGWorker implements TalkerIf{
  name: string;
  zahl: number;
  stringBauer(d: string): void {
    console.log("NMGWorker");
  }
}

let myinstance: TalkerIf = new NMGWorker();
myinstance.name="Hallo";
console.log(myinstance.zahl);
```

In der Praxis hilft dies beispielsweise bei der Reduktion der Abhängigkeit zu Clouddiensten. Wer seinen Übersetzungsdiest vom Rest des Codes über ein sauberes Interface trennt, hat bei API-Änderungen weniger Aufwand.

Leider sind wir damit noch nicht am Ziel. Wer sein Interface in der Hauptdatei ablegt, züchtet sich über kurz oder lang einen neuen Monolithen heran. Erfreulicherweise hilft uns Visual Studio auch hier. Platzieren Sie den Cursor im ersten Schritt auf dem Namen `NMGWorker`, um die weiter oben verwendete Glühbirne auf den Bildschirm zu holen. Wählen Sie daraufhin die Option *Move to new file*, um Visual Studio zur Durchführung des Refactorings zu animieren. Die Klasse findet sich fortan jedenfalls in der Datei `NMGWorker.ts`. Verschieben Sie auch die zweite Implementierungsklasse in ihr eigenes File.

Interessanterweise funktionieren sowohl IntelliSense als auch Kompilation nach wie vor. Probleme gibt es erst, wenn Sie das Programm zur Ausführung freigeben. Die ausgegebene Fehlermeldung informiert uns darüber, dass Node.js die Klasse `NMGWorker` nicht finden konnte:

```
tamhan@tamhan-thinkpad:~/tsspace/tsspace$ node out/
hallo.js
/home/tamhan/tsspace/tsspace/out/hallo.js:1
```

```
(function (exports, require, module, __filename,
__dirname) { var myinstance = new NMGWorker();
ReferenceError: NMGWorker is not defined
```

Ein Blick in *hallo.js* informiert uns über die Ursache des Problems. Wer das Projekt im vorliegenden Zustand transpiliert, sieht das folgende Resultat:

```
var myinstance = new NMGWorker();
myinstance.name = "Hallo";
console.log(myinstance.zahl);
//# sourceMappingURL=hallo.js.map
```

Zur Lösung des Problems müssen wir *hallo.ts* anpassen. Die Import-Befehle übernehmen ins lokale Dateisystem zeigende Pfade. Eine Besonderheit des TypeScript-Compilers ist, dass er bei der Suche im Verzeichnis mit der Restdatei das Voranstellen von *./* voraussetzt:

```
import {NMGWorker} from "./NMGWorker";
export interface TalkerIf{
    name: string;
```

TypeScript-Importe unterscheiden sich insofern von C++ und Co, als sie den Inhalt der Datei nicht stupide in das andere File kopieren. Der Transpiler führt stattdessen eine Merge-Operation durch, die nur die mit dem Export-Statement markierten Teile der Datei ansprechbar macht. Da wir das Interface *TalkerIf* in den Klassen verwenden wollen, müssen wir es mit einem *export*-Statement versehen.

NMGWorker.ts benötigt ebenfalls Aufmerksamkeit. Neben dem Export-Deklarator vor der Deklaration der Klasse müssen wir ein weiteres Import-Statement platzieren – das in *Haloo.ts* befindliche Interface ist sonst nicht wirklich sichtbar:

```
import {TalkerIf} from "./hallo";
export class NMGWorker implements
TalkerIf {
    name: string;
    zahl: number;
```

Die Verwendung der *exports*-Deklaration ist übrigens unbedingt erforderlich. Unterbleibt sie, so zeigt tsc bei der Kompilation einen Berechtigungsfehler an:

```
tamhan@tamhan-thinkpad:~/tsspace/
tsspace$ tsc
NMGWorker.ts:1:9 - error TS2305:
Module './hallo' has no exported
member 'TalkerIf'.
```

An dieser Stelle sei angemerkt, dass das wilde Auslagern von Klassen nicht immer

zielführend ist. Achten Sie darauf, die Anzahl der Import- und Exportanweisungen zu minimieren – wer statt fünf kurzen Interfaces zehn Deklarationen erzeugt, gewinnt nur wenig.

Konstruktoren und Voreinstellungen

Unser Default-Konstruktor ist in praktischen Systemen insofern subideal, als er das Erzeugen von nicht parametrisierten Objekten erlaubt. Stellen Sie sich beispielsweise eine Funktion vor, die einen String in *name* erwartet. Schreibt der Entwickler diesen nicht ein, so folgt seltsames Verhalten auf den Fuß. Der einfachste Weg zur Umgehung des Problems ist, die Klasse *NMGLage* um einen Konstruktor zu erweitern:

```
export class NMGWorker implements TalkerIf {
    name: string;
    zahl: number;
    constructor(name: string, zahl: number)
    {
        this.name = name;
        this.zahl = zahl;
    }
}
```

Die hier gezeigte Struktur erklärt anschaulich, warum Microsoft das Voranstellen von *this* bei Zugriffen auf Klassenmitglieder zwingend voraussetzt. Namenskollisionen sind insbesondere in Konstruktoren ein höchst ärgerliches Problem. Zur Sicherstellung der Kompilierbarkeit müssen wir in *hallo.ts* die beiden Konstruktorparameter beleben. Da die Klasse nun einen expliziten Konstruktor mitbringt, ist die Defaultimplementierung deaktiviert:

```
let myinstance: TalkerIf = new
NMGWorker("Name", 22);
myinstance.name="Hallo";
```

In praktischem Code wünscht man sich mitunter Parameter als optional zu deklarieren. Dies ist in TypeScript kein Problem, die beiden folgenden Deklarationen sorgen für gültige Konstruktoren:

```
constructor(name: string="XYZ",
zahl: number=42)
constructor(name: string="XYZ",
zahl: number)
```

Wichtig ist, dass das Anliefern eines Default-Parameters das Anfügen des Fragezeichens verbietet. Dies ist insofern logisch, als ein mit einer Standardeinstellung ausgestatteter Parameter nicht unbedingt erforderlich ist. Im Internet findet man trotzdem immer wieder falschen Code, der zu Fehlermeldungen führt.

Verpflichtende Parameter müssen zudem unbedingt vor ihren optionalen ►



Debugging: Die Fehlerjagd kann beginnen (Bild 9)

Kollegen zu liegen kommen. Dies ist wichtig, weil der TypeScript-Interpreter bei bunt gemischten Funktionen nicht weiß, wo die angelieferten Werte unterzubringen sind.

TypeScript unterstützt auch Ableitungen. Zur Demonstration dieses Patterns möchte ich mit der folgenden, aus der Dokumentation entnommenen Klasse beginnen. Sie implementiert eine Methode samt Default-Parameter, die Informationen in die Kommandozeile ausgibt:

```
class Animal {
  move(distanceInMeters: number = 0) {
    console.log('Animal moved ${distanceInMeters}m.');
  }
}
```

Ableitungen erfolgen in TypeScript nach dem von C# bekannten Schema. Das `extends`-Schlüsselwort erlaubt dem Entwickler das Anmelden der zu verwendenden Basisklasse:

```
class Dog extends Animal {
  bark() {
    console.log('Woof! Woof!');
  }
}
```

Der Zugriff auf die Member erfolgt dann wie aus C# bekannt. Wer die in `Dog` implementierten Methoden verwenden möchte, darf dies nicht über eine Variable vom Typ `Animal` tun:

```
const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();
```

Neben Interfaces kennt TypeScript auch abstrakte Klassen, die den Implementator zur Erzeugung bestimmter Memberfunktionen zwingen können. Ein netter Aspekt von TypeScript ist, dass die abstrakte Klasse auch Methoden mitbringen darf.

C++ und Java erleichtern Entwickler die Arbeit mit Datenstrukturen durch das Konzept der Generics. Eine auf Generics basierende Klasse wird bei der Entwicklung mit einem Parameter ausgestattet, dessen Typ erst im Rahmen der Nutzung festgelegt wird. Während Java und Co. Programmierer im Allgemeinen zur Generic-Festlegung auf Klassenebene zwingen, erlaubt TypeScript auch die Realisierung generischer Funktionen:

```
function arbeiter<T>(arg: T): T {
  console.log(typeof arg);
  console.log('---');
  return arg;
}
```

In der eckigen Klammer erwartet TypeScript dabei einen String, der im Rest der Funktionsdeklaration anstelle einer Typdeklaration zum Einsatz kommt. Unsere Funktion `arbeiter`

Links zum Thema

- TypeScript-Compiler-Einstellungen
<https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>
- Typen der TypeScript-Runtime
<https://www.typescriptlang.org/docs/handbook/basic-types.html>
- TypeScript-Projektseite
<https://www.typescriptlang.org>
- CoffeeScript-Projektseite
<http://coffeescript.org>
- Node.js
<https://nodejs.org>

nimmt einen Parameter namens `T` entgegen und liefert auch eine `T`-Instanz an den Aufrufer zurück. Zudem geben wir den Namen des angelieferten Objekttyps an. Der `typeof`-Operator retourniert einen String mit dem Typnamen. Ein schönes Beispiel ist die Verwendung eines `TalkerIf`-Objekts. Es lässt sich folgendermaßen verwenden:

```
let myinstance: TalkerIf = new NMGWorker("Name", 22);
console.log(arbeiter<TalkerIf>(myinstance));
```

TypeScript ist auch hier zum Erkennen des Typs befähigt. Wer `arbeiter` mit einer streng typisierten Variablen aufruft, parametriert das Generic automatisch:

```
let myinstance: TalkerIf = new NMGWorker("Name", 22);
console.log(arbeiter(myinstance));
```

In beiden Fällen sieht die Ausgabe identisch aus. Generische Klassen funktionieren im Allgemeinen genauso. Die zu ihrer Realisierung und Verwendung notwendige Syntax sieht folgendermaßen aus:

```
class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;
}
let myGenericNumber = new GenericNumber<number>();
```

TypeScript erlaubt Entwicklern das Platzieren statischer Konstrukte in generischen Klassen. In diesem Fall weiß der Transpiler allerdings nicht, welcher Typ im statischen Teil zu verwenden ist – ein Problem, das Microsoft brachial löst. Der statische Teil einer generischen Klasse muss voll durchtypisiert sein – Unklarheiten sind somit ausgeschlossen.

Iteratoren und Syntactic Sugar

Kaum ein Entwickler von Programmiersprachen kann der Versuchung widerstehen, sein Werkzeug mit mehr oder we-

niger umfangreichen Komfortfunktionen auszustatten. Ob des immensen Umfangs von TypeScript kann ich Ihnen hier nur einen kleinen Ausschnitt präsentieren. Beginnen wir mit Iteratoren:

```
let list = [4, 5, 6];

for (let i in list) {
    console.log(i); // "0", "1", "2",
}

for (let i of list) {
    console.log(i); // "4", "5", "6"
}
```

Auf den ersten Blick sehen die beiden hier abgedruckten *for*-Schleifen komplett identisch aus – die Unterschiede zeigen sich bei der Programmausführung. For-In-Schleifen iterieren über die Indizes eines Felds, während For-Of-Schleifen die Inhalte zurückliefern.

Ein weiteres Problem sind Bibliotheksmethoden, die durch Strings parametriert werden. User können hier im Allgemeinen beliebige Parameter anliefern, die der Bibliothekscode dann irgendwie abweisen muss.

TypeScript löst dies durch eingeschränkte Stringtypen. Ein schönes Implementierungsbeispiel ist die folgende Funktion, die nur drei Easingarten entgegennimmt:

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
class UIElement {
    animate(dx: number, dy: number, easing: Easing) {
        if (easing === "ease-in") {
            // ...
        }
        else if (easing === "ease-out") {
        }
        else if (easing === "ease-in-out") {
        }
    }
}
```

Würde ein Nutzer sie aus TypeScript-Code heraus mit ungültigen Daten aufrufen, so verweigert tsc die Transpilation.

Fehlersuche mit TypeScript

Rigide Typisierung erschwert Entwicklern die missbräuchliche Verwendung von Programmfunctionen. Leider ist dies keine silberne Kugel – TypeScript-Code ist nicht vor logischen und sonstigen Fehlern gefeit. Wegen des Transpilations-Prozesses ist die Fehlersuche normalerweise schwierig, weil die Runtime und die in ihr enthaltenen Debugger-Funktionen ja nur das Transpilat zu sehen bekommen.

Unsere Version der besprochenen Konfigurationsdatei wies den Transpiler zur Erzeugung von *map*-Dateien an. Diese erlauben die Korrelation zwischen TypeScript und JavaScript; komfortables Debugging in Visual Studio setzt allerdings Handarbeit voraus.

Drücken Sie im ersten Schritt Strg+Umschalt+D, um in den Debugging-Modus zu wechseln. Er unterscheidet sich vom Regelbetrieb insofern, als sie auf der linken Seite des Bildschirms nun die in Bild 9 gezeigte Debugging-Toolbar angezeigt bekommen.

Auf der linken oberen Seite findet sich ein Zahnradsymbol – nach dem Anklicken blendet die IDE im ersten Schritt ein Fenster ein, in dem sie sich für die Node.js-Ausführungsumgebung entscheiden. Die IDE generiert daraufhin eine Datei namens *launch.json*, in der die von Node zu verwendenden Parameter unterkommen. Im Fall unseres Programms sieht das File folgendermaßen aus:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "Launch Program",
            "program": "${workspaceFolder}/hallo.ts",
            "preLaunchTask": "tsc: build - tsconfig.json",
            "outFiles": [
                "${workspaceFolder}/out/**/*.js"
            ]
        }
    ]
}
```

Klicken Sie danach auf den Launch Program-Link, der oben links am Bildschirm erscheint. Visual Studio führt daraufhin einen Run des Programms durch, die Ausgabe erscheint im Terminalfenster.

Zur eigentlichen Fehlersuche greifen sie dann auf die gewohnten Werkzeuge zurück – klicken Sie beispielsweise auf die Zeilennummern, um Breakpoints festzulegen.

Fazit

Außer Frage steht, dass TypeScript die JavaScript-Ausführungsumgebung um eine Vielzahl von mehr oder weniger komfortablen Syntaxerweiterungen bereichert – dank der Flexibilität des Transpilers stehen Objektorientierung und Co auch dann zur Verfügung, wenn der Browser dieses Feature nicht unterstützt.

Die Nutzung lohnt sich schon deshalb – dass von .net-Sprachen oder Java umsteigende Entwickler mit TypeScript schneller warm werden, sei ebenfalls angemerkt. ■



Tam Hanna

ist Autor, Trainer und Berater mit den Schwerpunkten Webentwicklung und Webtechnologien. Er lebt in der Slowakei und leitet dort die Firma Tamoggemon Holding k.s. Er bloggt sporadisch unter www.tamoggemon.com